Parallel-Log-Single-Compaction-Tree: Flash-Friendly Two-Level Key-Value Management in KVSSDs

Yen-Ting Chen[1,*], Ming-Chang Yang[2,†], Yuan-Hao Chang[3,‡] and Wei-Kuan Shih[4,*]
[*]Department of Computer Science, National Tsing Hua University, Hsinchu, Taiwan
[†]Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
[‡]Institute of Information Science, Academia Sinica, Taipei, Taiwan
[1]andy499415045@gmail.com, [2]mcyang@cse.cuhk.edu.hk, [3]johnson@iis.sinica.edu.tw, [4]wshih@cs.nthu.edu.tw

Abstract— Log-Structured Merge-Tree (LSM-tree) based key-value store applications have gained popularity due to their high write performance. To further pursue better performance for key-value applications, various researches were conducted by adopting different architectures of flash devices, such as key-value solid-state drives (KVSSDs). However, since LSM-trees were originally designed based on the architecture of hard disk drives (HDDs), true potential of SSDs can not be well exploited without re-designing the management strategy. In this work, we propose Parallel-Log-Single-Compaction-Tree (PLSC-tree), which is a two-level and flash-friendly key-value management strategy specially tailored for KVSSDs. In particular, the first layer takes advantage of the massive internal parallelism of SSDs for maximizing the write performance via logging, while the second layer is designed to alleviate the internal recycling (i.e., compaction) overheads of flash devices for ultimately optimizing the performance on managing key-value pairs. A series of experiments were conducted based on a well-known SSD simulator with realistic workloads, and the results are very encouraging.

I. Introduction

Log-Structured Merge-Tree (LSM-tree) [11] key-value store applications have gained growing attentions due to their high write performance on data storage. An LSM-tree batches and writes key-value pairs to the storage device sequentially. It manages key-value pairs in multiple levels of files. Under this trend, researches are devoted to improving the performance of this kind of key-value application based on different architectures of flash devices, such as traditional solid-state drive (SSD), open-channel SSD and key-value SSD (KVSSD). However, most of these existing designs did not resolve the severe write amplification caused by managing key-value pairs in multiple levels of files. Thus, it is necessary to fundamentally re-design the structure of LSM-trees and cooperate with the characteristics of SSDs for ultimately improving the performance. This work is thereby inspired by the urgent need on proposing a new key-value management strategy, which not only can make good use of the internal parallelism but also can minimize the internal recycling overheads of key-value flash storage device.

Persistent key-value stores play an important role in large-scale and data-driven applications due to their efficient insertions and lookups with simple operational interface. For those write-intensive applications, LSM-trees such as Google's BigTable [2] and LevelDB [9] are considered to be the first choice and are designed based on the characteristics of hard-disk drives (HDDs). LSM-trees batch key-value pairs in the memory of the host system and write them to the storage device sequentially to achieve high write performance. Furthermore, in order to provide efficient lookups, LSM-trees read, sort, and write key-value pairs in the background to keep each of them as unique as possible. Consequently, additional sequential reads and writes are performed and causes severe read and write amplification. However, random I/Os of HDDs are much slower than sequential ones. The performance impact of random I/Os are severer than that of write amplifications caused by the LSM-trees. Therefore, it is a good deal to do such a trade-off for improving the performance of HDDs.

In order to achieve a higher degree of performance, people start to consider using flash storage devices (e.g., solid state drives) as the main storage. A flash storage device is usually composed of multiple NAND flash chips, which are connected into multiple channels. Each chip consists of dies and each die consists of planes. There are usually thousands of blocks in a plane and hundreds of pages in a block. Due to the hardware constraint, a flash page is the basic unit of read/write operations. Owing to the *write-once property*, flash pages that have been programmed/written can not be overwritten unless its residing block is erased first, where a flash block is the basic unit of erase operations. Thus, updates to data of flash pages are usually stored into other free pages rather than the original pages (referred to as *out-place update*). Because of the aforementioned hardware properties, an embedded management software called *flash translation layer (FTL)* is usually implemented to tackle the *address translation*, which maps the logical block addresses (LBA) referenced by the host system to the physical page addresses (PPA) in the flash memory. In addition, another crucial mechanism called *garbage collection* is also implemented in FTL to recycle the space of invalid pages so as to release the storage space occupied by the invalid data. On the other hand, based on the chip organization of flash storage device, four levels of parallelism (i.e., channel-level, chip-level, die-level and plane-level) and advanced commands (e.g., multi-plane operations [1]) are widely utilized by FTL for improving the access performance [6], [7].

Characteristics of SSDs are fundamentally different from HDDs and make the performance improvement restricted. Therefore, more and more researches are devoted to improving the performance of LSM-tree based key-value applications, which are built upon SSDs by investigating the bottleneck between host system and flash storage device [8], [13]. [8] proposed to separate keys from values and only keep keys

978-1-7281-4123-7/20/$31.00

sorted, so as to reduce the write amplification by avoiding unnecessary movement of values. [13] proposed to exploit the internal parallelism of flash devices by adopting open-channel SSDs as the main storage device to improve the performance. Notably, few researches even proposed a new SSD style, namely the key-value solid-state-drive (KVSSD), which supports key-value interface where key-value pairs can be directly read/written from/to the flash memory with key-value-specific APIs (i.e., put(), get() and delete() operations) [3], [4], [12]. In KVSSDs, the management of key-value pairs can be offloaded to the flash storage device and eliminate the unnecessary I/O stacks between the host system and the flash storage device for great performance improvement. [3], [4] first proposed the architecture of KVSSD and created different-sized storage units to store variable-sized value, in order to resolve the low space utilization problem caused by storing variable-sized key-value pairs into fixed-sized flash pages to achieve high performance. [12] proposed to integrate the LSM-trees into KVSSDs and use an in-RAM key-range tree to reduce the write amplification of LSM-trees.

However, the write amplification problem of LSM-tree based key-value applications mainly comes from processing multiple levels of files to reorganize the stored key-value pairs. Although existing designs have proposed different mechanisms to alleviate the write amplification, the degree of improvement over flash device performance is greatly restricted without re-designing the architecture of LSM-trees. Thus, this study proposes a two-level key-value management strategy, called Parallel-Log-Single-Compaction-Tree (PLSC-tree), which is designed based on the architecture of KVSSDs and cooperates with the characteristics of SSDs to ultimately achieve high read/write device performance. Inspired by LevelDB, *PLSC-tree* proposes to manage KVSSD in a two-level fashion, instead of multiple levels, to significantly reduce the write amplification. The first level is designed to maximize the write performance by leveraging the internal parallelism of SSDs. On the other hand, the second level is designed to alleviate the internal recycling overheads of flash device by reorganizing and storing key-value pairs with a smaller storage unit. Furthermore, to evaluate the capability of *PLSC-tree*, a series of experiments were conducted to compare with the LevelDB, which is one of the representative LSM-tree based key-value store management designs known for its high-performance. The evaluation results show that the read (resp. write) performance of the proposed *PLSC-tree* can achieve 28.6 (resp. 45.5) times faster than that of LevelDB under our test scenario.

The rest of this paper is organized as follow: Section II presents the background and motivation. Section III presents the new two-level key-value management strategy, namely *PLSC-tree*, to achieve high device performance for KVSSDs. Section IV presents the experimental results. Finally, we conclude this work in Section V.

II. Background and Motivation

LevelDB [9] is a widely used LSM-tree based persistent key-value store and is well-known for its high performance, especially the write performance. As shown in Figure 1, LevelDB is mainly composed of two in-memory sorted skip lists (i.e., *memtable* and *immutable memtable*) and seven levels of Sorted String Table (i.e., *SST*) files. When a key-value pair is going to be written, LevelDB initially appends it to an on-disk log file to prevent from data loss and then insert it to the *memtable*. Notably, the objective of *memtable* is to accumulate key-value pairs in memory of the host system to achieve sequential write for improving the write performance of hard disks. Once the *memtable* is full, a new *memtable* is generated to keep serving the incoming writes of key-value pairs and the previous *memtable* is converted to an *immutable memtable*. After that, key-value pairs in the *immutable memtable* are converted to data bytes (e.g., 2 MB) and flushed to the disk by a procedure called *minor compaction* to form an *SST*. In order to manage each *SST* in all levels, a file called *manifest* is in charge of recording the smallest and largest keys of each *SST* and is used to search key-value pair(s) as well. Besides, a *current* file is maintained to keep track of the newest version of *manifest* with considering the crash recovery.

In order to support efficient lookup on key-value pair(s), LevelDB recursively reorganizes *SSTs* in each level to its next level to recycle stale key-value pairs that have duplicated keys. In LevelDB, the total size of all *SSTs* in each level is limited and increased by level with a factor of ten, except for level 0. Once the total size of *SSTs* in a level is exceeded, a procedure called *major compaction* is triggered to reorganize and merge *SSTs* in this level to the next level. When *major compaction* is performed, one *SST* is chosen from a level and *SST(s)* with overlapped key range in the next level (e.g., level *i+1*) will be merged and generate new *SST(s)* to the next level (e.g., level *i+1*). This procedure will be continued until the total size of *SSTs* of each level is within their corresponding limitations. Notably, since *SSTs* in level 0 are directly flushed from the host system, keys might have multiple versions and the valid one is the last written one. Thus, except for level 0, the key ranges of *SSTs* in each level are non-overlapped. Consequently, maximum overhead to read a key is restricted to the total number of *SSTs* in level 0 and one *SST* from each of the other levels since only *SSTs* whose key range include the searched key will be checked.

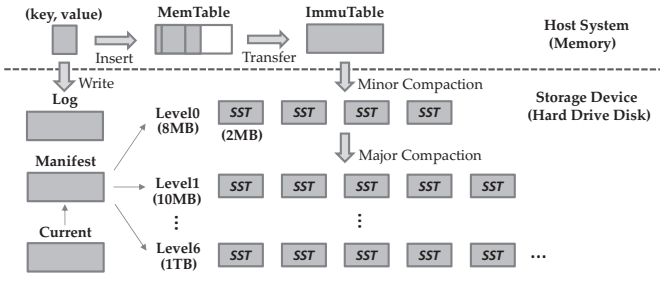


Fig. 1: Architecture of LevelDB.

With the growing demands for high-speed applications such as LevelDB, people start to consider flash storage devices (e.g., SSDs) as the main storage to achieve a higher performance. More and more researches were devoted to investigating the performance bottleneck between LSM-tree based key-value applications and flash storage devices. Among them, few researches have even proposed a brand new architecture of key-value-specific solid state drives (KVSSDs) to overcome

the gap of host system and SSDs. However, these researches did not fundamentally re-design the architecture of LSM-tree to meet the characteristics of flash memory. First, *SSDs have a large degree of internal parallelism that could be leveraged to maximize the read/ write performance* [6], [7]. Second, since flash memory does not support in-place updates, *multiple levels of compactions performed by LevelDB would incur severe write amplification and even trigger considerable numbers of additional reads/writes/erases*. Thus, this study presents a two-level key-value management strategy, in which each level cooperates with different characteristics of SSDs to achieve high performance.

III. PLSC-tree: Parallel-Log-Single-Compaction-Tree

A. Overview

In this section, a new *Parallel-Log-Single-Compaction-Tree* (referred to as *PLSC-tree*) is proposed to manage key-value pairs in a flash-friendly way to achieve high performance. As shown in Figure 2, our key idea is to manage the whole physical flash memory in a two-level fashion and each level considers to cooperate with different characteristics of flash devices. Notably, in order to manage these two logically separated storage spaces, *PLSC-tree* also introduces a new storage concept called *unsorted string table* (referred to as *UST*) to mange flash pages in groups of different sizes and is the basic allocated and reclaimed unit in *PLSC-tree*. Please note that the *USTs* are maintained by link-list because key-value pairs are batched and stored without sorting. The first level (i.e., level 0) is designed to maximize the write performance. *The number of flash pages in this level is aligned to the number of flash planes, so as to leverage the internal parallelism of the flash device (i.e., parallel-log)*. Thus, the number of flash pages of an *UST* in level 0 is always equal to the number of flash planes. On the other hand, the second level (i.e., level 1) is designed to alleviate the internal overheads of the recycling procedures, such as *compaction* and *garbage collection*, by *reorganizing and storing key-value pairs in an UST to a smaller storage unit (i.e., single-compaction)*. Thus, number of flash pages of an *UST* in level 1 could range from only one to the number of flash planes.

Besides, in order to provide efficient lookup for key-value pairs, *PLSC-tree* also proposes a *global index* to always keep track of their newest version by recording their residing *UST* and location. In *global index*, keys are separated into groups according to their length of byte and each group is a 256-entry hash table. Key-value pairs that are hashed to the same bucket in a group will be simply maintained by a link-list. Notably, the number of groups should depend on the maximum size of keys supported by the key-value application. Notably, *global index* is also designed to *greatly alleviate the write amplification introduced by compaction since duplicated keys do not need frequent merge to guarantee the lookup efficiency*. Last but not least, *PLSC-tree* is designed based on the architecture of the KVSSDs, which support key-value-specific APIs. Thus, a *string-byte converter* is responsible to convert key-value pairs between string and byte to overcome the gap of the key-value applications and the flash device. In order to show how these design goals are achieved, Section III-B and III-C elaborate the operations of KVSSD in level 0 (i.e., parallel-log) and level 1 (i.e., single-compaction), respectively.

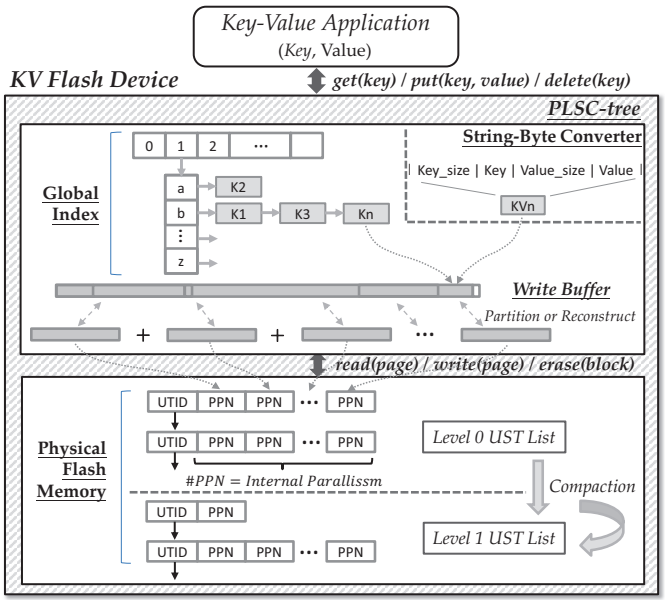


Fig. 2: Architecture of PLSC-tree.

B. Parallel-Log: Optimization of Write Performance

This section will present how the first level (i.e., level 0) logs key-value pairs sequentially and writes them to the physical flash memory in parallel for achieving high write performance, by demonstrating the procedures of the three most important key-value operations (i.e., put, get and delete).

Since the *put* operation is the starting point of all other operations, we elaborate the procedure of handling the *put(key, value)* operation first. As shown in Figure 3, when a *put("PLSC", "tree")* operation is initiated by a key-value application, the *string-byte converter* will be in charge of converting the key-value pair from strings to bytes. After that, converted data of bytes (e.g., *KV9*) of this newly put key-value pair is appended to the write buffer and a corresponding *kNode* is generated to keep track of it and maintained by a temporary link-list, called *KVbatchList*. Once the write buffer is full, data will be *partitioned and aligned to the flash page size and flushed to the flash planes in parallel*. Then, a level 0 *UST* identified by an unique ID (i.e., *UTID*) is formed to manage these flushed flash pages by their physical page number (i.e., *PPN*). Besides, in order to maintain the coherence between the physical flash memory and the *USTs* after *garbage collection* is performed, a *PPNtoUTID map* is used to maintain the correctness of mappings between *PPNs* and their residing *USTs*. After the flush operation is performed, all the *kNodes* in *KVbatchList* will be popped and inserted to the *global index*. Notably, each *kNode* records the necessary information to search for the corresponding key-value pair and its location, including the key of string, *UTID* of the residing *UST*, start byte address in the *UST* (i.e., *Offset*) and length of byte (i.e., *Len*). Notably, considering the comparison of strings is slow, each key shall provide a 128-bit signature to reduce the latency on searching a key.

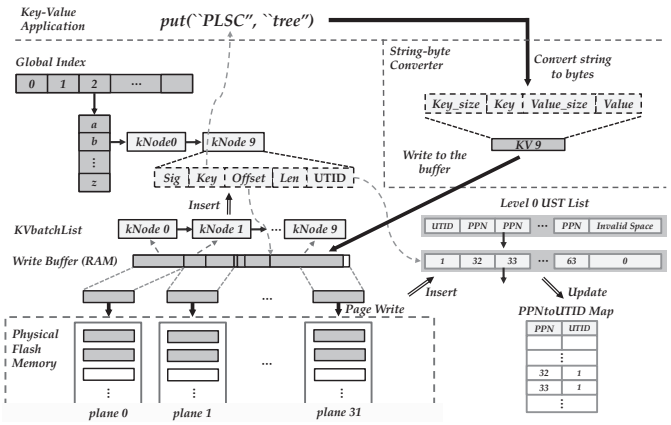


Fig. 3: Example of parallel log (i.e., *put* operation).

Furthermore, if the *put* operation is to put value to an existing key, then it is an update operation on this key. Since a key would only have one corresponding *kNode* in the *global index*, the older *kNode* will be updated to the newer one for maintaining the newest version of the updated key. Notably, before the *kNode* is updated, the invalidated storage space occupied by the stale key-value pair shall be updated to its corresponding *UST* in the *UST List*. Furthermore, in the case that the key is still in the write buffer, corresponding *kNode* in the *KVbatchList* will be updated to the newly put one and the following *kNodes* shall update their *Offset* as well.

The *get(key)* and *delete(key)* operations can be simply processed by searching the *KVbatchList* and *global index*. To perform the *get* operation, the *KVbatchList* is first looked up to find the given key. Once the corresponding *kNode* of the inquired key is found, data in the write buffer will be handed to the *string-byte converter* and the result will be returned after the conversion is completed. In the case that the inquired key is not found in the write buffer, the *global index* is first looked up to find the corresponding *kNode* of the given key. After that, indexes of the flash pages in *UST* that contain the inquired key-value pair can be located by calculating the *Offset* and *Len* in the *kNode*. Thus, corresponding flash pages can be read from the flash memory by their mapped *PPNs* in the *UST* and data is handed to the *string-byte converter* for returning the result. Similarly, to perform the *delete* operation, *KVbatchList* and *global index* are looked up to find the corresponding *kNode* of the given key. Once the *kNode* is found, it will be removed from the *KVbatchList* or *global index* and invalidated storage space of its residing *UST* will be updated if it is found in the *global index*.

C. Single-Compaction: Alleviation of Recycling Overheads

This section will present how key-value pairs are reorganized to the second level (i.e., level 1) and stored with a smaller *UST* to alleviate the internal overheads, by demonstrating the procedures of two important recycling processes (i.e., *compaction* and *garbage collection*).

When more and more *put* and *delete* operations are processed, invalid key-value pairs will gradually occupy the physical flash memory. Finally, all the free flash pages will be used up and cannot be allocated by any *UST* to store the newly put or updated key-value pairs. Thus, recycling mechanisms must be implemented in order to release the storage space of invalid key-value pairs. In *PLSC-tree*, *compaction* and *garbage collection* are proposed to recycle the storage space of invalid key-value pairs in cooperation. *Compaction* is responsible for logically recycling the invalid storage space by reorganizing the *USTs* in levels 0 and 1. On the other hand, *garbage collection* is responsible for physically releasing the storage space by reclaiming invalid flash pages to free pages.

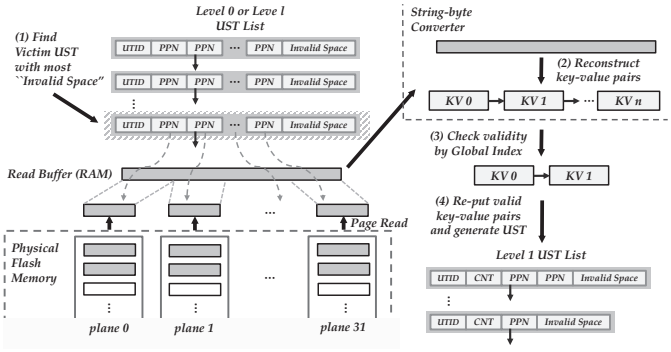


Fig. 4: Example of single *compaction*.

To prevent the free pages of flash memory from being used up, thresholds shall be defined to monitor the numbers of allocated flash pages of levels 0 and 1 respectively. Once the predefined threshold is reached, *compaction* is invoked and the process will not be stopped until the numbers of the allocated flash pages in levels 0 and 1 are both within their corresponding thresholds. Notably, since *USTs* in level 1 might be generated from level 0, *compaction* is first performed on level 0 and then on level 1. As shown in Figure 4, the first step is to find a worthwhile *UST* (referred to as *victim UST*) that could release the largest amount of invalid storage space. Thus, the corresponding *UST List* of the level that invoked *compaction* will be traversed to find the *UST* that owns the largest amount of invalid space as the *victim UST*. Then, all flash pages of this *victim UST* will be read from the physical flash memory to the read buffer by their *PPNs*. After that, data in the read buffer is handed to the *string-byte converter* to reconstruct the key-value pairs, which are temporarily maintained in a link-list. For each key-value pair, the validity will be checked by looking up the *global index* and valid ones will be re-put by following the same procedure of *put* operation (presented in Section III-B) to form new level 1 *UST*. Then, the *victim UST* will be discarded and all the flash pages allocated by it will be marked as invalid pages, which are waiting for *garbage collection* to reclaim them.

On the other hand, *garbage collection* will be invoked to reclaim invalid storage space when the number of remaining free pages in the flash memory is less than a predefined threshold. Notably, since *garbage collection* is a mature recycling mechanism, *PLSC-tree* is designed targeting to retain the original procedure as much as possible and only few modifications are required. When the *garbage collection* is invoked, the first step is to find a worthwhile flash block (referred to as *victim block*) that could release the largest amount of free storage space. That is, the flash block that owns the largest number of invalid flash pages would be chosen as the *victim block*. After that, each valid flash page in the

victim block will be copied to another available free page and this action is so-called *live-page copyings*. For each copied live-page, its corresponding *UST* will be visited by checking the *PPNtoUTID map* and the mapping shall be corrected by updating the old *PPN* to the new one. Finally, the whole *victim block* will be erased to release free storage space after all live-page copyings in the *victim block* are performed.

IV. Performance Evaluation

A. Experimental Setup

In this section, we evaluate the device performance and analyze the internal overheads (i.e., *compaction* and *garbage collection*) of the proposed PLSC-tree when it is applied to a key-value flash device. The experiment was conducted in a customized simulator that was modified from a well-known simulator called SSDSim [7]. SSDSim is a high-accuracy and configurable SSD-based simulator that supports advanced commands and evaluation on the device performance (e.g., read/write total response time). Thus, it has offered most of the basic framework for our key-value flash device. Based on the customized simulator, a 128 GB key-value flash device without warm-up was investigated, as shown in Table I. Other detailed configurations (e.g., the usage of advanced commands and threshold for *garbage collection*) follow the original settings recommended by SSDSim.

TABLE I: The evaluated key-value flash device.

Channel number	4	Chip number	8
Die number	2	Plane number	2
Block number	1024	Page number	256
Page size	16KB		

In addition, to demonstrate the effectiveness of the proposed PLSC-tree, LevelDB was also implemented into our customized simulator to represent the designs with the original LSM-tree [13], [12]. In our experiment, LevelDB is revised to be integrated in the key-value flash device and configurations, such as the size of each level, follow the standard design (e.g., 8 MB, 10 MB, 100 MB, etc.) as described in Section II. On the other hand, three representative key-value workloads (i.e., "small", "uniform", and "large") are used to evaluate the proposed PLSC-tree and the investigated LevelDB. Notably, the key sequence in the workloads is generated from the well-known Yahoo Cloud Serving Benchmark (YCSB) [5] based on the HBase database [10]. The total size of the put keys is 397 MB. Besides, in order to offer different value sizes for evaluation, the value for each key is generated in the three evaluated workloads. In the "small" and "large" distributions, the evaluated value sizes are all smaller or larger than the investigated flash-page size (i.e., 16 KB). The total sizes of the put values in the "small" and "large" distributions are of 7 GB and 30 GB, respectively. On the other hand, the evaluated value sizes are uniformly varied and the total size of put values is 14 GB in the "uniform" distribution. In our experiment, one million of *put* operations are first performed to test the write performance and are followed by one million of *get* operations to test the read performance under all workloads. Notably, to ease the following discussion, the *get* (resp. to *put*) operations will also be referred to as *read* (resp. to *write*) operations interchangeably.

B. Experimental Results

1) Performance: Figure 5 shows the total read response time introduced by LevelDB and PLSC-tree under all distributions, where the x-axis denotes the distribution of value sizes and the y-axis denotes the total response time to complete the *get* operations. As shown in the figure, PLSC-tree could achieve read performance for at least 19.4 times and at most 28.6 times faster than LevelDB under the "uniform" and "large" distribution, respectively. This is because LevelDB reads too many flash pages for a single *get* operation. In order to find a key, multiple *SSTs* from level 0 to levl 6 might be read from the flash memory. Furthermore, standard size of an *SST* file is 2 MB in LevelDB; this means that 128 flash pages will be read under our configuration. On the other hand, PLSC-tree can easily find a key by searching the corresponding *kNode* in *global index* and read a limited number of necessary flash pages.

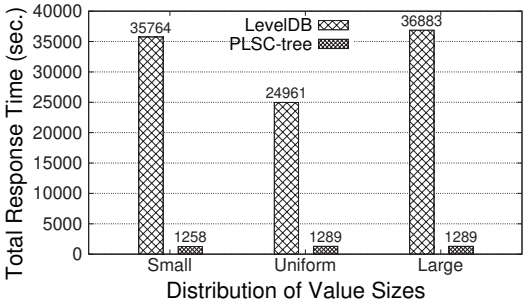
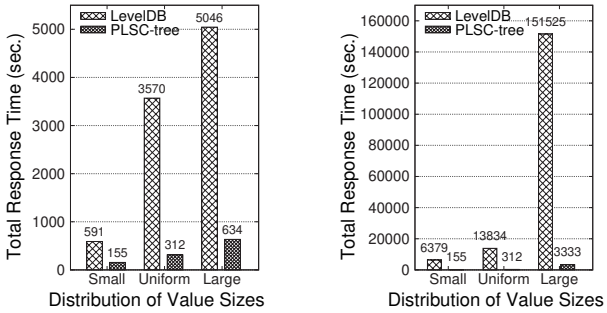


Fig. 5: Total read response time (of *get* operations).



(a) Excluding *compaction*. (b) Including *compaction*.

Fig. 6: Total write response time (of *put* operations).

Figure 6a and Figure 6b show the total write response time introduced by LevelDB and PLSC-tree under all distributions, where the x-axis denotes the distribution of value sizes and the y-axis denotes the total response time to complete the *put* operations. In Figure 6a, we first demonstrate the total write response time without considering the latency caused from the *compaction*, so as to show how the write performance varies while the internal parallelism of flash devices is considered. As shown in the figure, PLSC-tree could achieve write performance for at least 3.8 times and at most 11.4 times better than LevelDB under the "small" and "uniform" distribution, respectively. On the other hand, Figure 6b shows the total write

response time while the latency caused from the *compaction* is considered and should be regarded as the real response time of the flash device. As shown in the figure, the write performance of PLSC-TREE could achieve about 45.5 times faster than that of LEVELDB under the "large" distribution. From the test results, we can observe that the impact of *compaction* is tremendous, and such an impact could make the difference of write performance of these two designs become about 44 to 45 times. Thus, in order to further analyze the overheads of *compaction*, Section IV-B2 is introduced to break down the latency incurred by *compaction*.

2) Internal Overheads: Table IIa and Table IIb show the total read and write latency incurred by the *compaction* of the two investigated designs respectively. As shown in the Table IIa, read latency incurred by *compaction* of LEVEDB is about 25 times larger than that of PLSC-TREE in the worst case, namely the "large" distribution. On the other hand, Table IIb shows that the write latency incurred by *compaction* of LEVEDB is about 78.9 times larger than that of PLSC-TREE in the worst case, namely the "large" distribution. The reason is that LEVELDB performed a large number of *compactions* level by level to guarantee the efficiency of key lookups and introduce considerable flash-page reads/writes during the process. In contrast, PLSC-TREE performed *compactions* only when the number of free flash pages is below the predefined threshold. Besides, key-value pairs in level 1 are stored with smaller *USTs* and thus further alleviate the overhead of *compaction*.

TABLE II: Latency incurred by *compaction*.

Design	*Small*	*Uniform*	*Large*
LevelDB	5193 sec.	6684 sec.	51432 sec.
PLSC-tree	0 sec.	0 sec.	2065 sec.

(a) Read latency incurred by *compaction*.

Design	*Small*	*Uniform*	*Large*
LevelDB	595 sec.	3580 sec.	50048 sec.
PLSC-tree	0 sec.	0 sec.	634 sec.

(b) Write latency incurred by *compaction*.

Last, we analyze the overhead of *garbage collection* by presenting the number of performed erase and *copyback* (i.e., advanced command for accelerating live-page copyings) operations. Table IIIa shows the total numbers of erase operations of the two designs. As shown in the table, none of the erase operation is invoked in PLSC-TREE to release storage space while LEVELDB has performed considerable numbers of erase operations under "small" and "uniform" distributions. Under the "large" distribution, only few erase operations are performed in PLSC-TREE. The difference of the numbers of erase operations between the two designs becomes even larger. This phenomenon shows that LEVELDB incurs tremendous write amplification problem and largely increases the number of flash page writes. Similarly, since the performed number of *copyback* operations is strongly related to the number of performed erase operations, Table IIIb shows the same trend as Table IIIa.

TABLE III: Overhead incurred by *garbage collection*.

Design	*Small*	*Uniform*	*Large*
LevelDB	220832	326607	899967
PLSC-tree	0	0	3724

(a) Total operation counts of *Erase*.

Design	*Small*	*Uniform*	*Large*
LevelDB	2057925	12895077	102329794
PLSC-tree	0	0	152334

(b) Total operation counts of *Copyback*.

V. CONCLUSION

This paper proposes a new two-level key-value management strategy for key-value flash storage devices. The first level batches and writes key-value pairs to the flash planes in parallel to gain maximum write performance. The second level stores key-value pairs with smaller storage unit to alleviate the overheads of *compaction* and *garbage collection*. In particular, a new storage concept called *UST* is proposed to manage the flash pages in the two levels with different sizes and cooperate with the *global index* to support efficient lookups. The results show that the proposed design could achieve 28.6 times and 45.5 times better than the well-known LevelDB under our test scenario, in terms of read and write performance respectively.

REFERENCES

[1] A. R. Abdurrab, T. Xie, and W. Wang. Dloop: A flash translation layer exploiting plane-level parallelism. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 908–918, May 2013.

[2] 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 Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008.

[3] Y. Chen, M. Yang, Y. Chang, T. Chen, H. Wei, and W. Shih. Kvftl: Optimization of storage space utilization for key-value-specific flash storage devices. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 584–590, Jan 2017.

[4] Yen-Ting Chen, Ming-Chang Yang, Yuan-Hao Chang, Tseng-Yi Chen, Hsin-Wen Wei, and Wei-Kuan Shih. Co-optimizing storage space utilization and performance for key-value solid state drives. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, PP:1–1, 02 2018.

[5] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[6] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, June 2013.

[7] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren. Exploring and exploiting the multilevel parallelism inside ssds for improved performance and endurance. *IEEE Transactions on Computers*, 62(6):1141–1155, June 2013.

[8] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Wisckey: Separating keys from values in ssd-conscious storage. In *FAST*, pages 133–148, Santa Clara, CA, February 2016.

[9] Sanjay Ghemawat and Jeff Dean. Leveldb.

[10] The Apache Software Foundation. Apache HBase repository. https://hbase.apache.org/.

[11] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[12] S. Wu, K. Lin, and L. Chang. Kvssd: Close integration of lsm trees and flash translation layer for write-efficient kv store. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 563–568, March 2018.

[13] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. Flashkv: Accelerating kv performance with open-channel ssds. *ACM Trans. Embed. Comput. Syst.*, 16(5s):139:1–139:19, September 2017.