LLSM: A Lifetime-Aware Wear-Leveling for LSM-Tree on NAND Flash Memory

Dharamjeet, *Student Member, IEEE*, Yi-Shen Chen, *Member, IEEE*, Tseng-Yi Chen, *Member, IEEE*, Yuan-Hung Kuan, and Yuan-Hao Chang, *Senior Member, IEEE*

***Abstract*—The advancement of nonvolatile memory (NVM) technology reduces the cost-per-unit of solid-state drives (SSDs). Flash memory-based SSDs have become ubiquitous because they provide better performance and energy efficiency than hard disk drives. However, it suffers from wear-out problems caused by the out-of-place updates that limit its lifetime. Log-structured merge tree (LSM-tree) is a level-based data structure that is widely used in many database systems because it eliminates the random write operations to the storage devices. By transferring the random write operations into sequential write operations, the write performance of hard disk drives can be improved. However, LSM-tree is not efficient for SSDs because it is not aware of the access characteristics of flash memory. Moreover, the level-based indexing strategy of the LSM-tree significantly shortens the lifetime of SSDs because the data must be frequently updated due to the compaction operations between different levels. In contrast to many previous works that focus on alleviating the write amplification on SSDs for the database systems implemented by LSM-tree, we propose LLSM, a lifetime-aware wear-leveling for LSM-tree on NAND flash memory with open-channel SSD. By considering the data access frequency of the LSM-tree between different levels, LLSM rethinks the block allocation strategy during the compaction to evenly erase all the blocks of SSD storage devices, prolonging the SSD lifetime. Moreover, a proactive swapping strategy is designed to reorganize the data blocks for resolving the potential wear-leveling issues caused by the behaviors of the LSM-tree. The extensive experiments show that the results of lifetime improvement are encouraging.**

***Index Terms*—Endurance, lifetime, log-structured merge tree (LSM-tree), NAND flash memory, wear-leveling.**

I. Introduction

In the era of big data, many emerging Internet technologies, such as cloud computing, search engine, and social media have grown up and increased the amount of data rapidly [1], [2]. Therefore, the storage devices must be scaled up to handle such data in an efficient way. To this end, several database systems and key-value stores techniques have been developed to store and manage the unstructured data with structuring methods. The log-structured merge tree (LSM-tree) is one of the most popular index structures in the implementation of key-value stores. It transfers the random write operations into sequential write operations to reduce the seek time of hard disk drives. To accelerate the read operations, the LSM-tree stores and sorts the data in a hierarchical structure with different levels. When an LSM-tree level reaches its maximum capacity, all of the data in this level will be merged to the next level and sorted without overlapping ranges, referred to as "compaction." The compaction will be recursively executed until all levels do not reach their limits. Because LSM-tree can significantly improves the performance of database systems, several commercial available systems are implemented with LSM-tree, such as Bigtable [3], Dynamo [4] at Amazon, LevelDB [5] at Google, RocksDB [6] and Cassandra [7] at Facebook, HBase [8] at Apache, and PNUTS [9] at Yahoo.

With the rapid development of nonvolatile memory (NVM) technology [10], [11], [12], the cost-per-unit of flash memory-based solid-state drives (SSDs) is reduced. Because the SSD provides lower access latency and higher bandwidth than hard disk drives, it has become the mainstream technology to replace hard disk drives. In recent years, the density of SSD has gradually increased due to advanced chip manufacturing like multilevel-cell SSD (MLC SSD) [13], triple-level-cell SSD (TLC SSD) [14], and 3-D NAND SSD [15], so that more and more large-scale storage systems adopt SSDs to be their primary storage devices. However, SSDs still have limitations because of the property of erase-before-write and limited program/erase cycles. To conquer these problems, the flash translation layers (FTLs) [16] build bridges between the software and hardware layers for SSDs that maps the logical block address to the physical block address. Based on FTLs, the garbage collection [17] and wear-leveling [18], [19], [20], [21] techniques are developed to resolve the wear-out problems of SSDs. The garbage collection is triggered by the need for free space to reclaim the invalid blocks, and the wear-leveling is designed to evenly erase the valid blocks of SSD. Both of them are used to prolong the SSD lifetime.

Although LSM-tree can improve the access performance by eliminating the random write operations to sequential write operations for hard disk drives, it is not efficient for flash memory-based SSDs. Because the SSD devices cannot be aware of the behaviors of the LSM-tree and the LSM-tree

Manuscript received 2 August 2022; accepted 2 August 2022. Date of publication 9 August 2022; date of current version 24 October 2022. This work was supported in part by the Academia Sinica under Grant AS-IA-111-M01, Grant AS-GCS-110-08, and Grant AS-CDA-107-M05; and in part by the National Science and Technology Council under Grant 111-2223-E-001-001, Grant 111-2221-E-001-013-MY3, Grant 111-2628-E-008-002-MY3, and Grant 111-2221-E-008-058-MY3. This article was presented in the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2022 and appears as part of the ESWEEK-TCAD special issue. This article was recommended by Associate Editor A. K. Coskun. *(Corresponding authors: Yuan-Hao Chang; Tseng-Yi Chen.)*

Dharamjeet, Yi-Shen Chen, Yuan-Hung Kuan, and Yuan-Hao Chang are with the Institute of Information Science, Academia Sinica, Taipei 11529, Taiwan (e-mail: dharamjeet@iis.sinica.edu.tw; yschen195@iis.sinica.edu.tw; csyhkuan@gmail.com; johnson@iis.sinica.edu.tw).

Tseng-Yi Chen is with the Department of Computer Science and Information Engineering, National Central University, Taoyuan 32001, Taiwan (e-mail: tychen@g.ncu.edu.tw).

Digital Object Identifier 10.1109/TCAD.2022.3197542

is originally designed for hard disk drives, the unawareness between them not only fails to facilitate further optimizations but also induces more overheads. To be specific, due to the data structure of the LSM-tree, the data is stored in different levels, and the sizes of lower levels are much smaller than the upper levels. As a result, LSM-tree frequently merges and sorts the data from the lower levels to the upper levels, which causes uneven wear-out on SSD. In other words, the SSD physical blocks allocated to the lower levels of the LSM-tree will be erased more frequently than those allocated to the upper levels. Moreover, the LSM-tree is an out-of-place update structure that stores updates in new locations. It means that it always appends new data instead of modifying old data while updating the key-value stores. Consequently, the old data will be marked as invalid and wait for the garbage collection that sacrifices the read performance and space utilization of SSD [22].

Many efforts have been made to reduce the write amplification caused by the compaction of LSM-tree. The write amplification for the LSM-tree-based key-value store on SSD is mainly caused by the partial page updates and cascading updates during the compaction. It will affect the performance of the key-value store and seriously degrade the SSD lifetime. LOCS [23] exposes the internal flash channels to the applications with open-channel SSD (OCSSD) that enables multithreaded I/O access and modifies the scheduling and dispatching policies to improve the I/O throughput of SSD. LSM-trie [24] is a key-value storage system that stores the data in a trie structure (or a prefix tree) and does the compaction with a cryptographic hash function that ensures the key range is unique and nonoverlapping, thus improving the read/write throughput of SSD. FlashKV [25] accelerates the performance of the key-value store by limiting the number of flash channels used in compaction when it meets read-intensive workloads and proactively recycles the flash blocks containing the deleted files. sLSM-tree [26] is a skiplist-based LSM-tree that provides the ordered sequence of values for fast index searching by skipping the keys which are far away from the desired one. LWC-tree [27] employs a lightweight compaction method to mitigate I/O amplification by appending data in overlapped tables and merging their metadata simultaneously. KVSSD [28] integrates the LSM-tree and the FTL by remapping the metadata pages to key-value pages to implement copy-free compaction of LSM-tree, so that the write amplification is reduced and the efficiency of garbage collection is also improved. Although the previous work reduces the amount of data stored on flash memory and effectively handles the write amplification, they do not consider the endurance issues while applying LSM-tree-based key-value store on SSDs.

In contrast to the previous work, we propose LLSM, a lifetime-aware wear-leveling for LSM-tree on NAND flash memory with OCSSD. LLSM aims to prolong the lifetime of SSDs by jointly considering the access characteristics of SSD storage devices and the compaction of LSM-tree. The design of LLSM consists of two parts: 1) a lifetime-aware wear-leveling for LSM-tree that allocates the data ready to be stored in the lower levels to the blocks with less erase count during the compaction and 2) a proactive block swapping strategy for SSD that exchanges the data in the old and young blocks according to the predefined threshold to avoid the wear-out problems. The extensive experiments indicate that LLSM can evenly distribute the erase count of SSD blocks to prolong the lifetime of flash-based SSDs.

The main contributions of this work are summarized as follows.

1) We identify the endurance issue caused by the compaction of LSM-tree on NAND flash memory and propose LLSM, a lifetime-aware wear-leveling for LSM-tree on flash memory-based SSDs with the consideration of both the access characteristics of SSD storage devices and the compaction of LSM-tree to efficiently prolong the SSD lifetime.
2) We observe the potential wear-out problems while the LSM-tree-based key-value store is running on the SSDs for a long time and propose a block swapping strategy for SSD to proactively prevent these problems by the designed swapping threshold.
3) We conduct a series of experiments to evaluate the capability of LLSM in terms of lifetime improvement and compare LLSM with different well-known wear-leveling methods on various benchmarks. The experimental results show that LLSM significantly prolongs the SSD lifetime.

The remainder of this article is organized as follows. Section II presents the background and motivation of this work. Section III describes the design of the proposed lifetime-aware wear-leveling for LSM-tree on flash memory-based SSDs. In Section IV, the capability of LLSM is evaluated, and Section V concludes this work.

II. Background and Motivation

In order to clearly illustrate the details of our design, we first introduce the LSM-tree-based key-value store in detail. Then the background of SSD technology is presented, especially for the OCSSD, which is used to implement the proposed LLSM in this work. Finally, we clearly illustrate our motivation with an example of LSM-tree-based key-value store on SSDs.

A. *Log-Structured Merge Tree*

To improve the write performance of data management systems, key-value stores [29], [30], [31] are widely used in modern data centers. Currently, most key-value stores are implemented based on the LSM-tree [32], [33], [34] because it can provide low-cost indexing for HDDs. LSM-tree applies the log structure to append new data that transforms the random write operations into sequential write operations by adopting merge sort to keep the files in sequential order. Because random write operations are almost two orders of magnitude slower than sequential write operations on HDDs, they can provide better write performance than other index structures (e.g., B-tree).

Fig. 1 illustrates the data structure and basic operations of the LSM-tree. When a key-value pair is generated from the application layer, it first be sorted according to its key and inserted into an in-memory buffer "MeMTable" to update the old values. Once the MeMTable reaches its capacity limit, the

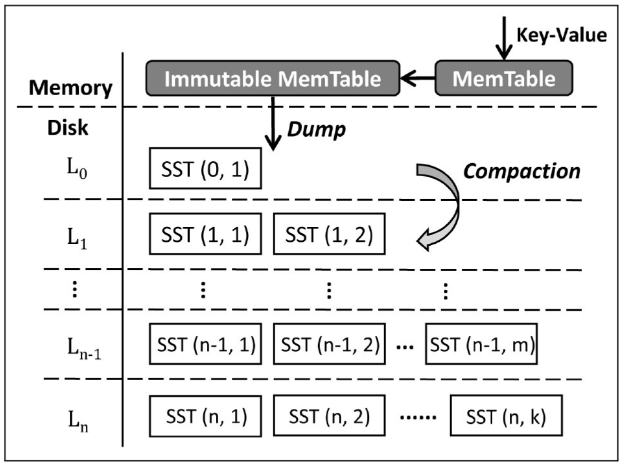


Fig. 1. Data structure and architecture of LSM-tree.

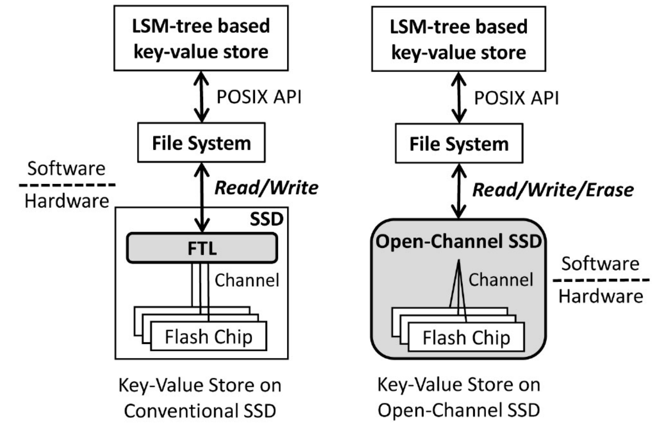


Fig. 2. Architectures of LSM-based key-value store on the conventional and OCSSD.

key-value pairs in the MeMTable will be converted to a read-only "Immutable MemTable." Then the Immutable MemTable will be dumped to the disk and stored with a sorted string table "SSTable" (referred to as "SST" in this figure) in level 0 layer (i.e., L_0). The SSTable is immutable and includes a number of key-value pairs. These key-value pairs can be indexed by the metadata of SSTable and be located by binary searching. The LSM-tree organizes SSTables in a hierarchical structure with different levels (i.e., L_0, L_1, etc.) and always dumps new SSTables to the top level L_0. The maximum number of SSTables in each level is limited and increases exponentially from L_0 to L_n. When the number of SSTables exceeds the limitation of a level. The *compaction* operation will be triggered to sort and merge the involved SSTables to the next level and then delete the old SSTables. For example, if the number of SSTables in L_0 is full, LSM-tree will apply compaction to select the SSTables in L_0 as the victim according to their overlapped key-value pairs. Then it performs merge sort and writes the new disjoint SSTables to the next level L_1. Finally, the invalid key-value pairs will be discarded.

Because the LSM-tree always dumps the incoming key-value pairs to the top level and performs compactions to sort and merge the SSTables from the L_i to L_{i+1} level, the lower levels will be frequently updated. As a result, it raises an endurance issue when applying LSM-tree to the database systems that operate on flash memory-based SSDs. Because the physical blocks of SSDs that store the data of lower levels will be erased more frequently, we observe that such behaviors not only significantly reduce the SSD lifetime but also cause potential uneven wear-out on SSDs.

B. Open-Channel SSD

In recent years, NAND flash memory such as SSD has become the mainstream storage device for large-scale storage systems because of its low access latency and high throughput, compared to the HDD. However, the performance and endurance of SSDs are limited by their inherent property of erase-before-write and limited program/erase cycles. Most of the SSDs are equipped with an FTL to implement the address mapping from the logical block address to the physical block address. Based on FTL, SSDs are able to exploit garbage collection, block management, and over-provisioning to alleviate the performance and endurance problems. However, these additional functionalities cause I/O bottleneck and write amplification [35], [36], [37], which significantly reduce the performance and lifetime of SSDs.

OCSSD [38], [39], [40] is introduced to be a promising solution for SSD to address the critical issues discussed above. It exposes the hardware information to the host that allows the users can directly control the inner layer of SSD [41]. Fig. 2 shows the architecture of LSM-based key-value store on conventional SSD (on the left side) and OCSSD (on the right side). OCSSD shares the responsibility of SSD management between the host and SSD. Unlike conventional SSDs, the OCSSD enables host-level FTL to eliminate the redundant management between FTL and the file system. The flash chips are connected to the controller in parallel via channels, and each channel contains multiple parallel units to process the I/O requests independently. Although OCSSD removes the information gap between the host and SSD, we observed that it still induces potential endurance problems because it was originally designed for general-purpose storage systems. In other words, it is not aware of the behaviors of LSM-based key-value store (e.g., compaction), so the SSD lifetime will be significantly reduced. In this work, we implement our LLSM with OCSSD and make it aware of the behaviors of LSM-tree.

C. Motivation

The lifetime is a critical issue for flash memory-based SSDs, especially for large-scale database systems, which usually adopt LSM-tree key-value store for indexing. In order to prolong the endurance of SSDs and avoid wearing out the flash blocks prematurely, many previous works focus on developing *wear-leveling* [42], [43] for NAND flash SSDs. Wear-leveling aims to evenly distribute the writes to the flash blocks, so that the flash blocks can endure more erases without putting too many erase burdens on certain flash blocks. However, to the best of our knowledge, existing wear-leveling strategies do not consider the behaviors of the LSM-tree, so they

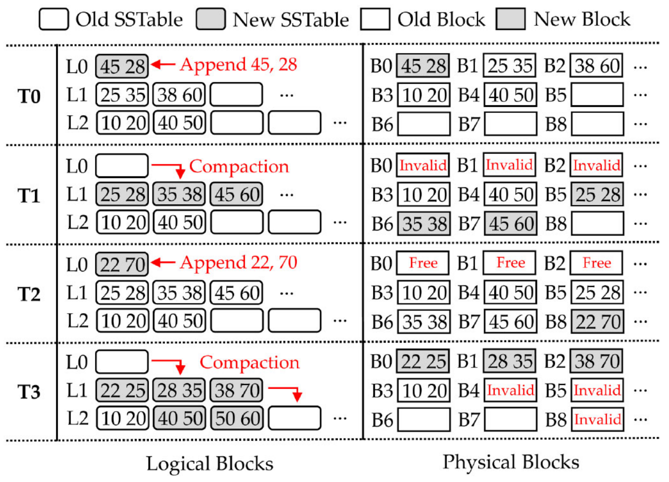


Fig. 3. Motivation example of LSM-tree-based key-value store on NAND flash memory.

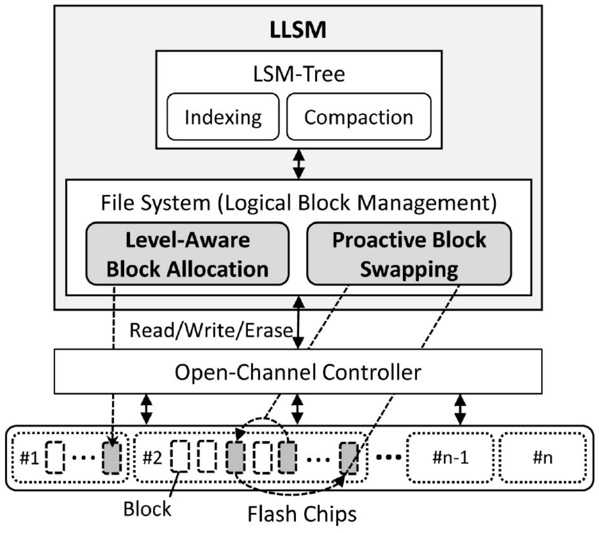


Fig. 4. System architecture of the proposed LLSM.

fail to solve the endurance issues caused by LSM-tree-based key-value stores on SSDs.

Fig. 3 depicts how the LSM-tree affects the lifetime of NAND flash memory. To simplify the explanation, we assume that an SSTable is stored in a single block. The left side shows the logical blocks of SSD for the LSM-tree, and the right side represents the physical blocks of SSD. As shown in the figure, at time $T0$, an SSTable with a key-value pair (45, 28) is appended to the level $L0$ of the LSM-tree. At this time, block B0 is allocated to store this SSTable. After that, because $L0$ reaches its capacity of SSTables, LSM-tree performs compaction to merge the SSTable to $L1$ at time $T1$. Meanwhile, the block allocation of SSD assigns blocks $B5$, $B6$, and $B7$ to store the new SSTables, and the blocks $B0$, $B1$, and $B2$ become invalid; these blocks will be reclaimed by the garbage collection later. Next, at time $T2$, a new SSTable with the key-value pair (22, 70) is again appended to the $L0$. This SSTable is stored in block $B8$, and then the blocks $B0$, $B1$, and $B2$ have been reclaimed by the garbage collection and become the free blocks. Moreover, because $L0$ has reached its capacity of SSTables, the compaction is triggered to merge the SSTable from $L0$ to $L1$. However, the $L1$ also reaches its maximum number of SSTables; thus, the LSM-tree performs the compaction again for merging the SSTables in $L0$ and $L1$ to the next level $L2$. Since the blocks $B0$, $B1$, and $B2$ are free blocks, SSD allocates them to store the merged SSTables. In this case, the compaction operations of the LSM-tree frequently erase the blocks $B0$, $B1$, and $B2$. It implies that some physical blocks like $B0$, $B1$, and $B2$ will quickly wear out due to the compaction of LSM-tree, causing uneven wear-out. Furthermore, we also observe that if the NAND flash memory is active for a long time, the blocks stored in the SSTables at upper levels are rarely accessed by LSM-tree because it always appends new SSTables to the lower levels, aggravating the wear-out problems.

This work is inspired by the need to prevent uneven wear-out while applying LSM-tree on flash memory-based SSDs. We observe that the LSM-tree will significantly shorten the lifespan of SSDs due to the unawareness between the LSM-tree indexing strategy and the access characteristics of flash memory. Our goal is to design a lifetime-aware wear-leveling for LSM-tree on flash memory-based SSDs to prolong the SSD lifetime. To achieve this goal, we put our efforts into integrating the LSM-tree with the proposed block allocation and swapping strategy. The technical problem falls on: 1) how to evenly distribute the data in the LSM-tree to the physical blocks of SSD and 2) how to intelligently swap the old and young blocks of SSD to prevent the uneven wear-out caused by the LSM-tree.

III. LLSM: Lifetime-Aware Wear-Leveling for LSM-Tree on NAND Flash Memory

A. Overview

In order to prolong the SSD lifetime for LSM-tree-based key-value store, we propose LLSM, a lifetime-aware wear-leveling for LSM-tree on NAND flash memory with OCSSD. LLSM rethinks the block allocation strategy of SSD-based file systems. In contrast to the related work that randomly distributes data to the blocks for general-purpose storage systems, LLSM aims to erase the blocks evenly considering the hierarchical data structure of the LSM-tree. Fig. 4 shows the system architecture of the proposed LLSM. As shown in the figure, LLSM includes the LSM-tree and realizes its novel data placement policy via the file system. It is deployed and integrated into the OCSSD to enable physical block management. Because the LSM-tree employs the log structure to arrange the key-value pairs in the persistent storage, it is feasible to allocate the data upon the flash chips. To make the LLSM solve the unawareness between the LSM-tree indexing strategy and the access characteristics of flash memory, we design a level-aware block allocation in Section III-B to properly allocate the data in the LSM-tree to the memory blocks according to the data hotness based on the LSM-tree levels. Moreover, a proactive block swapping is introduced in Section III-C to intelligently swap the old and young blocks to prevent uneven wear-out among the memory blocks in terms of long-term usage.

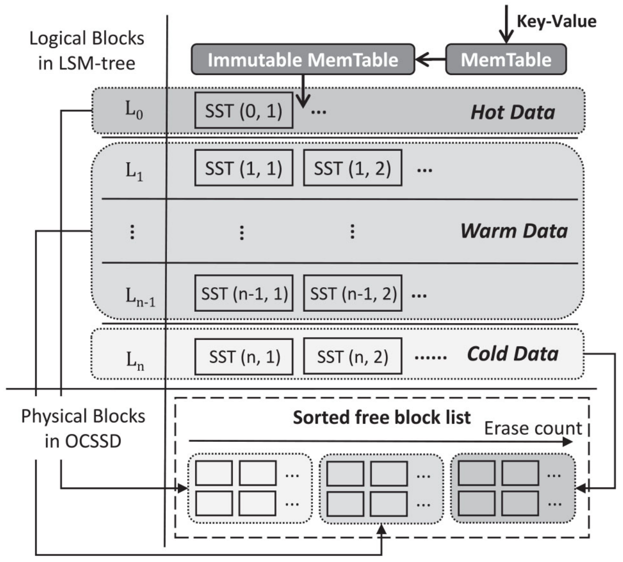


Fig. 5. Level-aware block allocation.

B. Level-Aware Block Allocation

Due to the limited endurance of flash memory, the data placement of the incoming data to the SSD is critical. Because the LSM-tree stores data in a log-structured way that always writes the data to the lower levels and the capacity of lower levels is much smaller than the upper levels, it frequently triggers compaction for the lower levels that induce a lot of update/write requests to certain flash blocks. Such observations motivate us to design a level-aware block allocation strategy that aims to allocate the SSTable of the LSM-tree to the flash blocks according to its data hotness. In the conventional SSDs, memory controllers manage the data placement via the FTL, so that it is invisible between the logical blocks in LSM-tree and the physical blocks in SSD. As a result, even though the FTL applies many additional functionalities like block management and garbage collection in the file system, it still fails to prolong the SSD lifetime because it does not have any information about the storage device at the hardware level. In this work, we implement our LLSM on OCSSD to manage the raw flash space at the application level.

Fig. 5 illustrates the proposed level-aware block allocation. As mentioned in Section II-A, the data structure of the LSM-tree is a hierarchy with different levels. When a key-value pair appends to the LSM-tree, it will be sorted and inserted into an in-memory buffer MeMTable. If MeMTable reaches its threshold, it will be converted to a read-only Immutable MemTable and be dumped to the L_0 of LSM-tree. After that, the dumped key-value pairs are maintained in a sorted string table SSTable. When the number of SSTables in L_0 reaches its capacity limit, LSM-tree performs the compaction to merge and sort the involved SSTables containing overlapping keys to the next level (i.e., L_1) in the background. It will be recursively performed until all of the SSTables in each level do not exceed the capacity limits of the corresponding levels. LLSM collects the SSTables to the size of a flash block, so as to store the SSTables with a single block. The main idea of the proposed level-aware block allocation is allocating the SSTables based on their data hotness in LSM-tree. We classify the data hotness into three categories: 1) hot; 2) warm; and 3) cold, which are referred to as "hot data," "warm data," and "cold data," respectively. Because L_0 is the lowest level of the LSM-tree, the update/write requests applied to these levels are much more than the other upper levels (e.g., L_1 to L_n). Therefore, the SSTables in L_0 are regarded as the hot data in the LSM-tree. When the number of levels in the LSM-tree is n, the update frequencies of L_1 to L_{n-1} are usually higher than L_0 but lower than L_n. Consequently, the SSTables between L_1 to L_{n-1} are referred to as the warm data. The cold data has the lowest update frequency in the LSM-tree, and the SSTables in L_n are classified as the cold data.

To prevent the flash blocks be worn out prematurely, LLSM maintains a sorted free block list (SFBlist) in the SSD via open-channel architecture. The SFBlist includes the free physical blocks in the SSD and sorts them based on their erase counts in forward sequences. Note that the erase count of each flash block can be easily retrieved and updated while erasing a block, so that the overhead of obtaining the erase count is negligible. In the SFBlist, the free physical blocks with similar erase counts are grouped together for block allocation. Because the hot data will be frequently updated, which significantly increases the P/E (program-erase) cycles of flash blocks, LLSM assigns the hot data to be stored in the physical blocks with a minimum erase count. On the contrary, the cold data is rarely updated after it has been stored in the flash block, so LLSM will store it in the physical blocks with maximum erase count.

To properly allocate the warm data, LLSM designs an allocation algorithm and makes the decision by an allocation criterion. The allocation criterion of warm data α can be determined by 1, where L_i is the index of the level at which the warm data currently resides, L_t is the total number of levels in the LSM-tree, $\mathrm{EC}_{\min}$, and $\mathrm{EC}_{\max}$ are the minimum and maximum of the erase count among the free blocks, respectively. By allocating the free blocks for warm data according to the designed allocation criterion, the erase count of each block can be evenly distributed, so as to prolong the SSD lifetime as much as possible

$$\alpha = \frac{L_i}{L_t} \times \frac{\mathrm{EC}_{\min} + \mathrm{EC}_{\max}}{2}. \tag{1}$$

Algorithm 1 is the proposed level-aware block allocation for warm data. It will be triggered when the LSM-tree performs compaction to merge and sort the involved SSTables with overlapping keys to generate new SSTables in the next level, which is referred to as warm data in our block allocation strategy. First, LLSM calculates the allocation criterion α for warm data, and then check the free blocks in SFBlist with α via is_available() (steps 1 and 2). If there exists a free block with α erase count, LLSM sets the erase count of block $\mathrm{EC}_{\mathrm{block}}$ to α and allocates warm data to this block via Block_Allocation() (steps 3 and 4). In most cases, LLSM maintains the SFBlist that provides enough free blocks for warm data. However, if there is no free block with α erase count in SFBlist, LLSM starts to search the free blocks for warm data in the next level. It is implemented by increasing the L_i by one and updating α to check the free blocks in the SFBlist, so that the warm data is ready to be stored in level

Algorithm 1: Level-Aware Block Allocation for Warm Data

```
Input: Sorted free block list SFBlist
Input: Warm data $D_{warm}$
Input: Index of the level for warm data $L_i$
Input: Total number of levels $L_t$
Input: Minimum of erase count $EC_{min}$
Input: Maximum of erase count $EC_{max}$
/* Allocate the free block for warm data */
1  $\alpha = (L_i/L_t) * ((EC_{min} + EC_{max})/2)$;
2  if SFBlist.is_available($\alpha$) then
3      $EC_{block} = \alpha$;
4      Block_Allocation($D_{warm}$, $EC_{block}$);
5  else
       /* Search the free block for warm data in the next level */
6      while $L_i + 1 <= L_t$ do
7          $L_i + 1$;
8          $\alpha = (L_i/L_t) * ((EC_{min} + EC_{max})/2)$;
9          if SFBlist.is_available($\alpha$) then
10             $EC_{block} = \alpha$;
11             Block_Allocation($D_{warm}$, $EC_{block}$);
12     Block_Allocation($D_{warm}$, $EC_{max}$);
```

$L_i + 1$; it will be recursively performed until $L_i + 1 > L_t$ (steps 5–9). Once there has a free block with the corresponding erase count α, LLSM stores warm data in the free block with erase count $\mathrm{EC_{block}}$ (steps 10 and 11). Otherwise, LLSM assigns the free block in SFBlist with maximum erase count (i.e., $\mathrm{EC_{max}}$) to store warm data because the search range is over the total number L_t of levels in the LSM-tree (step 12).

Conventional NAND flash memory applies random or dynamic wear-leveling to alleviate the uneven wear-out on SSD. The former randomly chooses a free block to store the incoming data, regardless of its hotness or the erase count of the target block; the latter considers the erase count of the flash block and dynamically allocates a free block with the lowest erase count. Nevertheless, both of them are designed for general-purpose storage systems that do not consider the behavior of the LSM-tree. In contrast to the random and dynamic wear-leveling, the proposed LLSM rethinks the block allocation of SSD to evenly distribute the incoming data on the flash blocks. By applying level-aware block allocation on SSDs, the behaviors of LSM-tree-based key-value store can be perceived to prolong the SSD lifetime.

C. Proactive Block Swapping

Although LLSM applies level-aware block allocation to alleviate the uneven wear-out of SSD by considering the data hotness of LSM-tree and the erase count of flash blocks during compaction, there still exists some unresolved problems in terms of long-term usage. Based on our observations, about 70% of the key-value pairs are stored in the upper levels of the LSM-tree. These key-value pairs are updated rarely and gradually become cold data in the storage device over time. The P/E cycles of the flash blocks with the cold data will be much lower than the other data (i.e., hot and warm data), causing a potential uneven wear-out if the LSM-tree-based key-value store is active on SSD for a long time. Note that the uneven wear-out caused by cold data cannot be tackled by the proposed level-aware block allocation in Section III-B because the compaction is rarely triggered at these upper levels. To this end, we propose proactive block swapping to tackle the potential uneven wear-out on SSD caused by cold data in terms of long-term usage.

The main design idea of the proposed block swapping strategy is to proactively change the allocation of hot and cold data among the physical blocks by considering their erase counts in the background. Fig. 6(a) shows the overview of physical blocks in the SFBlist maintained by LLSM. SFBlist includes the free physical blocks in the SSD via the open-channel architecture and sorts the free blocks based on their erase counts in forward sequences. The erase count of each block can be easily retrieved and updated while erasing a block with negligible performance overhead. The free physical blocks with similar erase counts are grouped together to allocate hot/warm/cold data, which are marked by dark grey, grey, and light grey, respectively.

Fig. 6(b) shows the procedure of the proposed proactive block swapping based on the SFBlist in Fig. 6(a). It will be triggered after a physical block in SFBlist is allocated to store the SSTables by level-aware block allocation. LLSM first picks up a block with maximum erase count in the SFBlist (i.e., B_{0i} with 120 erase times) and then compares its erase count with the swapping threshold β (Step 1). The swapping threshold β is a metric that represents the median erase count in the sorted free block list (i.e., SFBlist), which helps LLSM to determine the timing of triggering the proactive block swapping. It is defined in (2), where $\mathrm{EC_{min}}$ and $\mathrm{EC_{max}}$ are the minimum and maximum erase count in the SFBlist, respectively. If the erase count of B_{0i} is greater than β, LLSM chooses a block with a minimum erase count in the SFBlist (i.e., B_{20} with 12 erase times) for swapping (steps 2 and 3). Meanwhile, an empty block with a heavy erase count in the SFBlist (i.e., B_{0n} with 113 erase times) is used to place the data in B_{20} (i.e., h) (steps 4 and 5). After that, the data in B_{0i} (i.e., c) is swapped to B_{20}, and the erase count of B_{20} is increased by one (Step 6). Finally, B_{0i} becomes an invalid block and waits to be reclaimed by the garbage collection. In this way, the uneven wear-out caused by cold data can be solved by the proposed proactive block swapping because it intelligently swaps the data in old and young blocks. To be specific, the cold data will not stay in the block with less erase count for a long time; the hot data no longer significantly increases the erase count of the block, which has been erased many times. Instead, the hot data is allocated to be stored in the block with a minimum erase count that further prolongs the entire SSD lifetime

$$\beta = \frac{\mathrm{EC_{min}} + \mathrm{EC_{max}}}{2} \tag{2}$$

$$\gamma = \beta \times \frac{E_{\mathrm{avail}}}{E_{\mathrm{avail}} + \beta}. \tag{3}$$

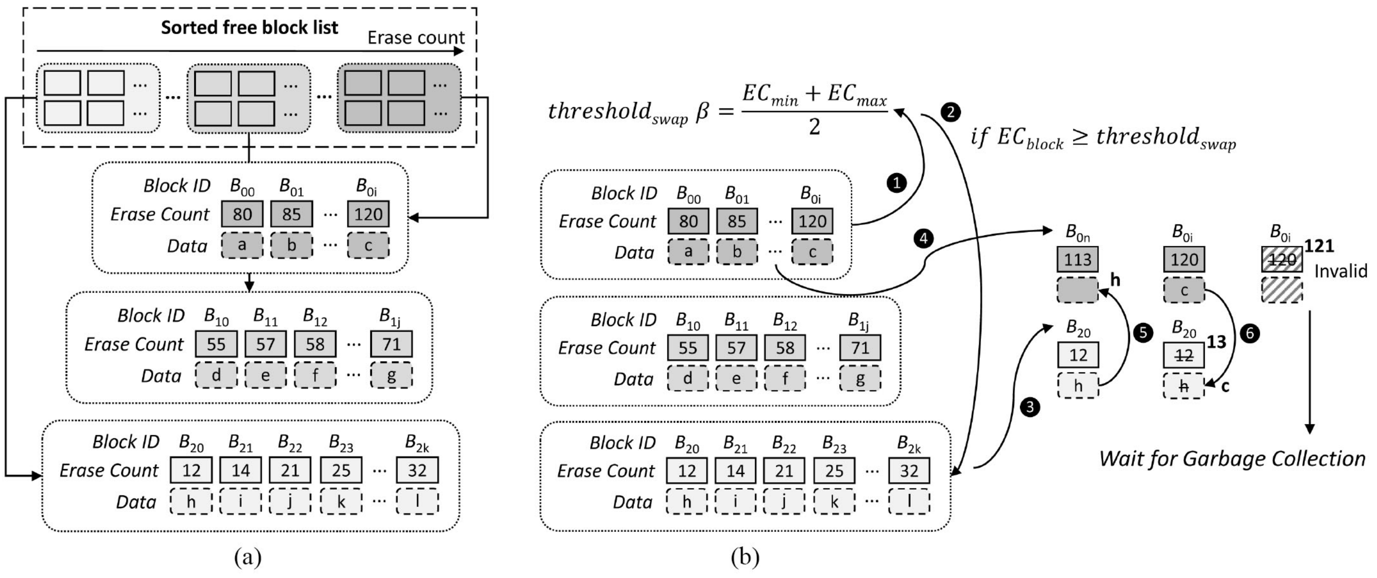

Sorted free block list
Erase count
Block ID
Erase Count
Data
threshold_swap β = (EC_min + EC_max)/2
if EC_block ≥ threshold_swap
h
13
c
121
Invalid
Wait for Garbage Collection
(a)
(b)

Fig. 6. Illustration of the proposed proactive block swapping. (a) Physical blocks in the SFBlist. (b) Procedure of proactive block swapping.

Considering the usage scenarios of SSDs, we do not need to apply the proactive block swapping with extreme strength at the beginning of the SSD lifetime; on the contrary, the strength of the proactive block swapping must be strengthened when the flash blocks are almost worn-out. To this end, we put more effort into designing (3) to enable the adaptive control for the proposed proactive block swapping by replacing β with γ, where E_{avail} is the percentage of available endurance of SSD. For example, if E_{avail} equals to 100 and β is 25, then γ is $25 \times (100/125) = 20$, which let γ be smaller than β that only decreases the swapping threshold by 20%. On the other hand, if β is increased to 60, then γ is $60 \times (100/160) = 37.5$, which lets γ be much smaller than β that significantly decreases the swapping threshold by 37.5%. In other words, this will significantly strengthen the strength of the proactive block swapping because it seriously decreases the swapping threshold from 60 to 37.5, incurring more block swappings. Furthermore, if the E_{avail} is less than 100 (i.e., the available endurance of SSD is less than 100%), the γ will become much smaller to further strengthen the strength of the proactive block swapping. With the intelligent design of (3), we can replace the swapping threshold β with the designed γ, so that the proposed proactive block swapping can adaptively adjust its strength by considering both the average erase count among all blocks and the current SSD lifetime.

IV. Performance Evaluation

A. Evaluation Metrics and Experimental Setup

This section describes a series of experiments conducted to evaluate the proposed lifetime-aware wear-leveling for LSM-tree on SSD (referred to as "LLSM") in terms of write count, first failure time, live page copying, and the distribution of erase count. To evaluate the performance and efficiency of wear-leveling with our LLSM, we implement two other wear-leveling for comparison: random wear-leveling (referred to as "Random WL") and dynamic wear-leveling (referred to as "Dynamic WL"). We first modify and employ sLSM-tree [26] as a tool to generate the LSM-tree-based key-value pairs from the investigated workloads and then feeds them into the FTL layer of OCSSD to implement the functionality of Random WL, Dynamic WL, and LLSM in terms of wear-leveling on SSD. When users want to store the data in the SSD, Random WL will randomly choose a free block to store the incoming data, regardless of its hotness or the erase count of the target block. In contrast, Dynamic WL considers the erase count of the flash block and dynamically allocates a free block with the lowest erase count for the incoming data. For the proposed LLSM, it considers both the behaviors of the LSM-tree and the access characteristics of SSD by applying level-aware block allocation and proactive block swapping to achieve even wear of the entire flash blocks. The experiments are conducted with a 16GB OCSSD. The SST table size is 92800 bytes, which is obtained by multiplying the capacity of buffer size and the size of each key-value pair (i.e., 800×116). We apply three wear-leveling (i.e., Random WL, Dynamic WL, and LLSM) on TPC-H [44] and Yahoo! Cloud Serving Benchmark (YCSB) [45]. TPC-H and YCSB are well-known open-source database benchmark suites, and they are often used to evaluate the performance of database management systems [46], [47]. TPC-H is able to generate update requests, and YCSB allows synthetically generating a series of insert requests to the storage systems. Besides TPC-H, six build-in distributions in YCSB are chosen for evaluating the performance of our LLSM: Zipfian, Uniform, Sequential, Hotspot, and Exponential. The detailed experimental setup is shown in Table I.

B. Experimental Results

1) Write Count: The main objective of the proposed lifetime-aware wear-leveling is to prolong the lifetime of SSD for LSM-tree-based key-value store. To quantify the SSD lifetime, we measure the number of writes (referred to as "write count" in this work) before any failure occurs on the SSD. To be specific, the write count represents the total number

TABLE I
EXPERIMENTAL SETUP

Benchmark	
TPC-H, Yahoo Cloud Serving Benchmark (YCSB)	
Wear-leveling methods	
Random wear-leveling	
Dynamic wear-leveling	
LLSM	
NAND flash memory-based SSD	
Capacity	16 GB
Number of flash chips	16
Number of blocks	6384
Size of each block	1024 KB
Number of pages	128
Page size	8 KB
LSM-tree based key-value stores	
Buffer capacity	800 B
Size of key-value pair	116 B
Runs per level	20
Number of runs per disk level	20

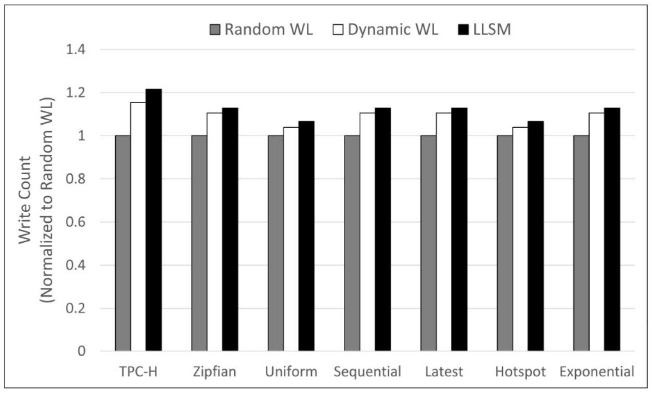


Fig. 7. Write count for TPC-H and the six different workloads of YCSB.

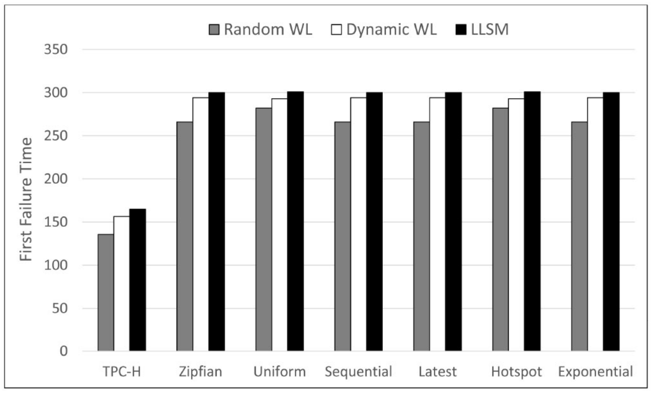
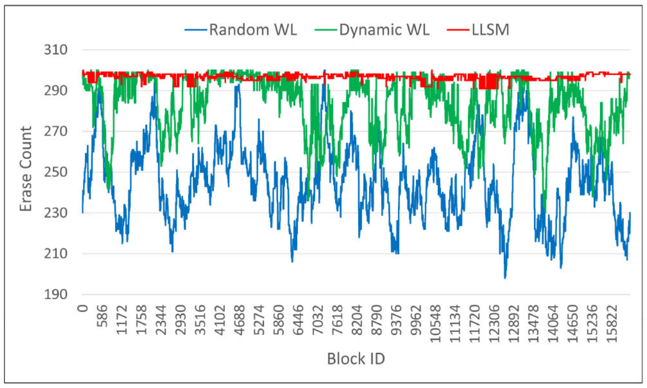


Fig. 8. First failure time for TPC-H and the six different workloads of YCSB.

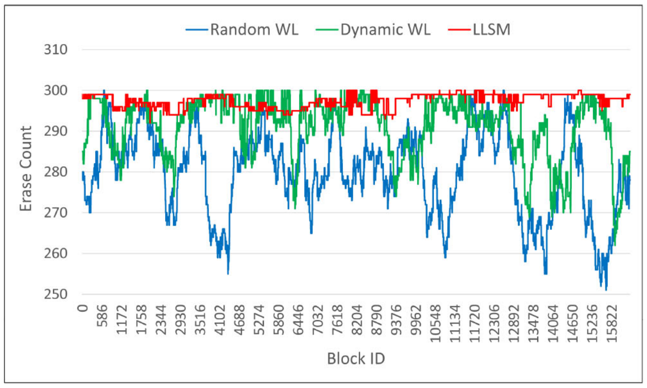
of writes until any flash block is worn-out. The more write counts endured by the SSD, the longer SSD lifetime is provided by the applied wear-leveling technique. Fig. 7 shows the write count for TPC-H and the six different workloads of YCSB with Random WL, Dynamic WL, and the proposed LLSM. The y-axis is normalized to 1 with respect to the write count of applying Random WL. The experimental results show that the proposed LLSM yields a lifetime improvement up to 21.7%, compared to that of Random WL. Moreover, by applying our LLSM, the SSD can endure more write counts up to 5.1%, compared to Dynamic WL. Because Random WL and Dynamic WL do not consider the behaviors of the LSM-tree, the flash blocks of SSD will be unevenly worn out, which significantly reduces the lifetime of SSD. In contrast, by applying LLSM, the SSD can endure more write counts until failure because it applies level-aware block allocation to properly allocate the physical blocks for the incoming data according to the hierarchical data structure of the LSM-tree. Moreover, a proactive block swapping is designed to proactively swap the young and old blocks in the background to prevent the flash blocks be worn out prematurely.

2) *First Failure Time:* The first failure time is defined as the rewind times of the investigated benchmarks. It represents how many times of data loading the SSD can afford until it meets the first failure. Each experiment is conducted by continuously loading the representative benchmark to the SSD that inserts millions of key-value pairs until any one of the flash blocks is failed to reserve the data. Therefore, the longer period of the first failure time represents that the SSD loads more data as the time passes. Fig. 8 presents the results of the first failure time for TPC-H and the six different workloads of YCSB with Random WL, Dynamic WL, and the proposed LLSM. It is observed that the proposed LLSM can let the SSD reserve the data for a longer time, compared to Random WL and Dynamic WL. By applying LLSM, the rewind times of the investigated benchmarks can be more than 300 times besides TPC-H. Note that the SSD only can afford about 266 rewind times when applying Random WL for Zipfian, Sequential, Latest, and Exponential; it indicates that the first failure time will be reduced by 11.6% when the SSD applies Random WL as its wear-leveling algorithm rather than applies LLSM. Because the proposed LLSM not only addresses the uneven wear-out caused by the behaviors of the LSM-tree but also proactively swaps the young and old blocks to prevent some flash blocks be worn out prematurely, the SSD can afford more data loading in the database systems.

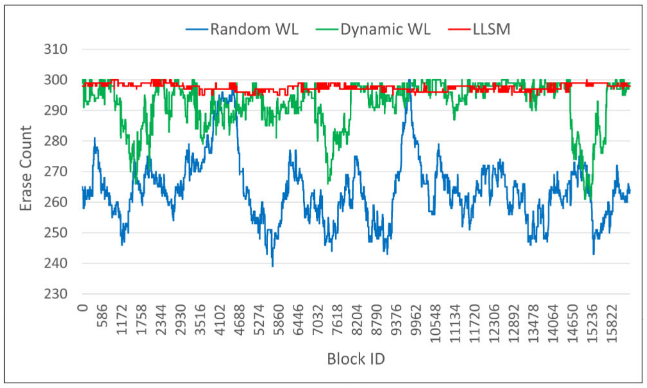
3) *Erase Count:* In order to evaluate the capability of LLSM in terms of wear-leveling, Fig. 9 shows the erase count distribution of the flash blocks when we apply the three wear-leveling algorithms (i.e., Random WL, Dynamic WL, and LLSM) to SSD on different benchmarks until the first failure. The erase count distribution not only helps us to understand the efficiency of each wear-leveling algorithm but also investigates the reason behind the results of the first failure time presented in Section IV-B2. To simplify the analysis, we only present three figures (i.e., TPC-H, Hotspot in YCSB, and Exponential in YCSB) here for the illustration because the result of Hotspot in YCSB is similar to the result of Uniform in YCSB, and the result of Exponential in YCSB is close to the results of Latest, Sequential, and Zipfian. Fig. 9(a) shows the erase count distribution of TPC-H after applying Random WL, Dynamic WL, and LLSM for wear-leveling. The x-axis denotes the block ID from 0 to 16383. The Random WL, Dynamic WL, and LLSM are represented by the blue, green, and red line, respectively. Note that we record the erase count distribution for each wear-leveling algorithm until the first failure. Therefore, the total erase count of flash blocks with Random WL is less than Dynamic WL and LLSM because of the premature failure of



(a)



(b)



(c)

Fig. 9. Erase count distribution of the flash blocks when applying different wear-leveling algorithms to SSD on different benchmarks until the first failure. (a) TPC-H. (b) Hotspot in YCSB. (c) Exponential in YCSB.

SSD. On the contrary, the total erase count of LLSM is greater than Random WL and Dynamic WL because LLSM makes the SSD endure more write counts. As shown in the figure, the erase counts of all flash blocks with LLSM are evenly distributed. By applying LLSM as the wear-leveling algorithm of SSD for TPC-H, the minimum erase count among the blocks is 291, while the maximum erase count among the blocks is 300; it is encouraging that LLSM controls the difference in erase count among the entire flash blocks no more than 3%. It is evident that LLSM outperforms Random WL and Dynamic WL in terms of wear-leveling. The similar erase count among the flash blocks also delay the first failure time while inserting a lot of key-value pairs into the LSM-tree-based key-value store. In contrast, the erase count of each block with Random WL is very different. The minimum erase count among the blocks with Random WL is 198, while the maximum erase count among the blocks is 300, causing about 34% difference between the two flash blocks. In other words, Random WL will induce skewed erase count distribution that may quickly wear out a single block to cause the failure of SSD. Although Dynamic WL dynamically considers the current erase count of the flash blocks and always erases the block with a minimum erase count first, the minimum erase count among the blocks is 230, while the maximum erase count among the blocks is 300; as a result, it still induces 23.3% difference between the two flash blocks that may prematurely wear out one of the flash blocks to cause the failure of SSD.

Fig. 9(b) and (c) show the erase count distribution of Hotspot and Exponential in YCSB, respectively, by applying Random WL, Dynamic WL, and LLSM as their wear-leveling algorithms. In summary, although the performance of LLSM for Hotspot is a little lower than TPC-H and Exponential, it still controls the difference in erase count among the entire flash blocks under 2.7%. On the other hand, Dynamic WL narrows the gap between the minimum and maximum erase count among all blocks to 12.7% and 13% for Hotspot and Exponential, respectively. Moreover, Random WL also narrows the gap between the minimum and maximum erase count among all blocks to 16.3% and 20.3% for Hotspot and Exponential, respectively. Note that the proposed LLSM performs better than Random WL and Dynamic WL in terms of wear-leveling by a factor of 4.9 and 7.7, respectively.

4) Live Page Copying: The number of live page copies is a metric to evaluate the overhead of our LLSM, compared to the Random WL and Dynamic WL. Live page copying is a necessary operation during the garbage collection on SSDs. When the garbage collection is triggered, it will find a block that contains one or more garbage pages and then read out the live (nongarbage) pages from that block. After writing out all of the live pages, the garbage collection starts to reclaim the entire block for subsequent use, improving the utilization of the storage space. In other words, more live page copies induce more overheads because the SSD has to spend more effort on copying and writing.

Fig. 10 shows the live page copying while applying Random WL, Dynamic WL, and LLSM as the wear-leveling algorithms on SSD for TPC-H and the six different workloads of YCSB. The y-axis is normalized to 1 with respect to the total number of live page copies of Random WL. Because Random WL randomly chooses a free block to store the incoming data without considering the data hotness or the erase count of the target block, it induces the fewest live page copies in all benchmarks. Unlike Random WL, Dynamic WL always allocates a free block with the lowest erase count to store the incoming data, thus incurring more live page copies than Random WL. Although LLSM incurs a greater number of live page copies than Random WL and Dynamic WL for TPC-H, Uniform, and Hotspot, the number of live page copies caused by LLSM is fewer than Dynamic WL for Zipfian, Sequential, Latest, and Exponential. The reason behind this phenomenon

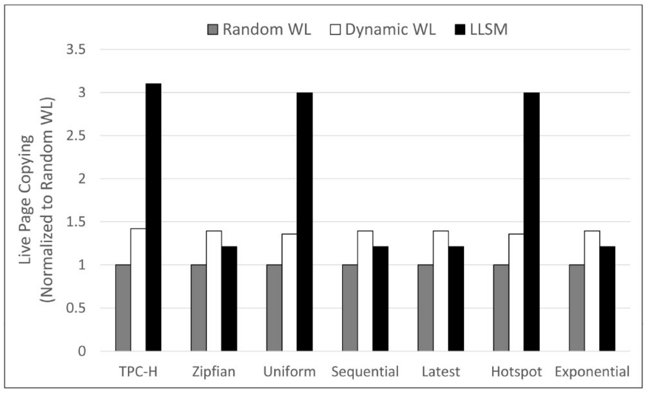


Fig. 10. Live page copying for TPC-H and the six different workloads of YCSB.

could be explained by the design of the allocation criterion in Section III-B. According to the allocation criterion α, the level-aware block allocation of LLSM will not always allocate the block with a minimum erase count to store the incoming data. In contrast, Dynamic WL always allocates the free block with the lowest erase count to store the incoming data, which usually contains a lot of live pages. To further investigate the large number of live page copies induced by LLSM, we analyze the total write throughput of the representative workloads. It is observed that the number of live page copies induced by LLSM is proportional to its capability of prolonging the SSD lifetime. Because the experiments of live page copies are measured until any flash block is worn-out, the results of total write throughput indicate that the SSD occurs a failure prematurely while applying Random/Dynamic WL as its wear-leveling strategy. As a result, the number of live page copies may be fewer than LLSM. In order words, LLSM makes SSD could endure more write requests than that of Random WL and Dynamic WL on these workloads.

V. Conclusion

In this work, we propose LLSM, a lifetime-aware wear-leveling for LSM-tree on NAND flash memory with OCSSD, to prolong the SSD lifetime. By considering both the behaviors of the LSM-tree and the access characteristics of SSD storage devices, LLSM efficiently solves the potential uneven wear-out caused by the LSM-tree-based key-value stores. LLSM applies level-aware block allocation to properly allocate the physical blocks of SSD according to the hierarchical data structure of the LSM-tree. Moreover, a proactive block swapping is designed to exchange the young and old blocks based on the level-aware block allocation to prevent the flash blocks be worn out prematurely. Extensive experimental results show that the proposed LLSM prolongs the SSD lifetime up to 21.7% and controls the difference in erase counts among the entire flash blocks within 2.7 to 3% in the investigated representative benchmarks.

References

[1] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *Proc. 12th ACM SIGMETRICS/PERFORMANCE Joint Int. Conf. Meas. Model. Comput. Syst.*, 2012, pp. 53–64.
[2] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan, "LinkBench: A database benchmark based on the facebook social graph," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2013, pp. 1185–1196.
[3] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.
[4] G. DeCandia *et al.*, "Dynamo: Amazon's highly available key-value store," *ACM SIGOPS Oper. Syst. Rev.*, vol. 41, no. 6, pp. 205–220, 2007.
[5] S. Ghemawat. "Fast key-value storage library." 2022. [Online]. Available: https://github.com/google/leveldb
[6] "A persistent key-value store for fast storage environments." 2022. [Online]. Available: http://rocksdb.org/
[7] "Open source NoSQL database." 2022. [Online]. Available: http://cassandra.apache.org/
[8] "Welcome to apache HBase." 2022. [Online]. Available: http://hbase.apache.org/
[9] B. F. Cooper *et al.*, "PNUTS: Yahoo!'s hosted data serving platform," *Proc. VLDB Endowment*, vol. 1, no. 2, pp. 1277–1288, 2008.
[10] Y.-S. Chen, C.-F. Wu, Y.-H. Chang, and T.-W. Kuo, "A write-friendly arithmetic coding scheme for achieving energy-efficient non-volatile memory systems," in *Proc. 26th Asia South Pac. Design Autom. Conf.*, 2021, pp. 633–638.
[11] C.-W. Chang, C.-Y. Yang, Y.-H. Chang, and T.-W. Kuo, "Booting time minimization for real-time embedded systems with non-volatile memory," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 847–859, Apr. 2013.
[12] Y.-S. Chen, Y.-H. Chang, and T.-W. Kuo, "Drift-tolerant coding to enhance the energy efficiency of multi-level-cell phase-change memory," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Design*, 2022, pp. 1–6.
[13] Y.-M. Chang, Y.-H. Chang, T.-W. Kuo, Y.-C. Li, and H.-P. Li, "Achieving SLC performance with MLC flash memory," in *Proc. 52nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, 2015, pp. 1–6.
[14] M.-C. Yang, Y.-H. Chang, C.-W. Tsao, and C.-Y. Liu, "Utilization-aware self-tuning design for TLC flash storage devices," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 10, pp. 3132–3144, Oct. 2016.
[15] T.-Y. Chen, Y.-H. Chang, C.-C. Ho, and S.-H. Chen, "Enabling sub-blocks erase management to boost the performance of 3D NAND flash memory," in *Proc. 53nd ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, 2016, pp. 1–6.
[16] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *J. Syst. Archit.*, vol. 55, nos. 5–6, pp. 332–343, 2009.
[17] M.-C. Yang, Y.-M. Chang, C.-W. Tsao, P.-C. Huang, Y.-H. Chang, and T.-W. Kuo, "Garbage collection and wear leveling for flash memory: Past and future," in *Proc. IEEE Int. Conf. Smart Comput.*, 2014, pp. 66–73.
[18] M.-C. Yang, Y.-H. Chang, C.-W. Tsao, and P.-C. Huang, "New ERA: New efficient reliability-aware wear leveling for endurance enhancement of flash storage devices," in *Proc. 50th ACM/EDAC/IEEE Design Autom. Conf. (DAC)*, 2013, pp. 1–6.
[19] Y.-M. Chang, P.-C. Hsiu, Y.-H. Chang, C.-H. Chen, T.-W. Kuo, and C.-Y. M. Wang, "Improving PCM endurance with a constant-cost wear leveling design," *ACM Trans. Design Autom. Electron. Syst.*, vol. 22, no. 1, pp. 1–27, 2016.
[20] S.-W. Cheng, Y.-H. Chang, T.-Y. Chen, Y.-F. Chang, H.-W. Wei, and W.-K. Shih, "Efficient warranty-aware wear leveling for embedded systems with PCM main memory," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 7, pp. 2535–2547, Jul. 2016.
[21] M.-C. Yang, Y.-H. Chang, T.-W. Kuo, and P.-C. Huang, "Capacity-independent address mapping for flash storage devices with explosively growing capacity," *IEEE Trans. Comput.*, vol. 65, no. 2, pp. 448–465, Feb. 2016.
[22] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, 2020.
[23] P. Wang *et al.*, "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 1–14.
[24] X. Wu, Y. Xu, Z. Shao, and S. Jiang, "LSM-trie: An LSM-tree-based ultra-large key-value store for small data items," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2015, pp. 71–82.
[25] J. Zhang, Y. Lu, J. Shu, and X. Qin, "FlashKV: Accelerating KV performance with open-channel SSDs," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 5s, pp. 1–19, 2017.
[26] A. Szanto, "The skiplist-based LSM tree," 2018, *arXiv:1809.03261*.
[27] T. Yao, J. Wan, P. Huang, X. He, F. Wu, and C. Xie, "Building efficient key-value stores via a lightweight compaction tree," *ACM Trans. Storage*, vol. 13, no. 4, pp. 1–28, 2017.

[28] S.-M. Wu, K.-H. Lin, and L.-P. Chang, "KVSSD: Close integration of LSM trees and flash translation layer for write-efficient KV store," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2018, pp. 563–568.

[29] B. Debnath, S. Sengupta, and J. Li, "FlashStore: High throughput persistent key-value store," *Proc. VLDB Endow.*, vol. 3, nos. 1–2, pp. 1414–1425, 2010.

[30] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky, "SILT: A memory-efficient, high-performance key-value store," in *Proc. 33rd ACM Symp. Oper. Syst. Principle*, 2011, pp. 1–13.

[31] R. Escriva, B. Wong, and E. G. Sirer, "HyperDex: A distributed, searchable key-value store," in *Proc. ACM SIGCOMM Conf. Appl. Technol. Archit. Protocols Comput. Commun.*, 2012, pp. 25–36.

[32] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil, "The log-structured merge-tree (LSM-tree)," *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[33] Y.-S. Kim, T. Kim, M. J. Carey, and C. Li, "A comparative study of log-structured merge-tree-based spatial indexes for big data," in *Proc. IEEE 33rd Int. Conf. Data Eng. (ICDE)*, 2017, pp. 147–150.

[34] H. Chen, C. Ruan, C. Li, X. Ma, and Y. Xu, "SpanDB: A fast, cost-effective LSM-tree based KV store on hybrid storage," in *Proc. 19th USENIX Conf. File Storage Technol. (FAST)*, 2021, pp. 17–32.

[35] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *Proc. 11th USENIX Conf. File Storage Technol. (FAST)*, 2013, pp. 257–270.

[36] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "A large-scale study of flash memory failures in the field," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 43, no. 1, pp. 177–190, 2015.

[37] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash reliability in production: The expected and the unexpected," in *Proc. 14th USENIX Conf. File Storage Technol. (FAST)*, 2016, pp. 67–80.

[38] H. Qin, D. Feng, W. Tong, J. Liu, and Y. Zhao, "QBLK: Towards fully exploiting the parallelism of open-channel SSDs," in *Proc. IEEE Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2019, pp. 1064–1069.

[39] Y. Lu, J. Shu, and J. Zhang, "Mitigating synchronous i/O overhead in file systems on open-channel SSDs," *ACM Trans. Storage*, vol. 15, no. 3, pp. 1–25, 2019.

[40] J. Chen, Y. Wang, A. C. Zhou, R. Mao, and T. Li, "PATCH: Process-variation-resilient space allocation for open-channel SSD with 3D flash," in *Proc. IEEE Design Autom. Test Europe Conf. Exhibit. (DATE)*, 2019, pp. 216–221.

[41] M. Bjørling, J. Gonzalez, and P. Bonnet, "LightNVM: The Linux open-channel SSD subsystem," in *Proc. 15th USENIX Conf. File Storage Technol. (FAST)*, 2017, pp. 359–374.

[42] M. Murugan and D. H. C. Du, "Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead," in *Proc. IEEE 27th Symp. Mass Storage Syst. Technol. (MSST)*, 2011, pp. 1–12.

[43] C. Wang and W.-F. Wong, "Observational wear leveling: An efficient algorithm for flash memory management," in *Proc. 49th Annu. Design Autom. Conf.*, 2012, pp. 235–242.

[44] "TPC-H vesion 2 and version 3." 2022. [Online]. Available: http://www.tpc.org/tpch/

[45] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.

[46] M. Barata, J. Bernardino, and P. Furtado, "YCSB and TPC-H: Big data and decision support benchmarks," in *Proc. IEEE Int. Congr. Big Data*, 2014, pp. 800–801.

[47] T. Chiba and T. Onodera, "Workload characterization and optimization of TPC-H queries on apache spark," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, 2016, pp. 112–121.

Dharamjeet (Student Member, IEEE) received the M.S. degree from the Department of Electrical and Computer Science Engineering, Chang Gung University, Taoyuan, Taiwan, in 2014, and the Ph.D. degree in Department of Electrical and Computer Science from a joint Ph.D. program between Academia Sinica, Taipei, Taiwan, and National Tsing Hua University, Hsinchu, Taiwan, in 2022.

He is currently a Principal Project Scientist with the Department of School of Information Technology, Indian Institute of Technology Delhi, New Delhi, India. His primary research focuses on data indexing and management, memory/storage systems and architectures, next-generation nonvolatile memory, and embedded systems.

Yi-Shen Chen (Member, IEEE) received the M.S. degree from the Department of Computer Science and Information Engineering, National Chung Cheng University, Minxiong, Taiwan, in 2016, and the Ph.D. degree from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, in 2021.

He is currently a Postdoctoral Scholar with the Institute of Information Science, Academia Sinica, Taipei. His primary research interests include emerging nonvolatile memories, memory/storage systems, and embedded systems.

Tseng-Yi Chen (Member, IEEE) received the Ph.D. degree in computer science from National Tsing Hua University, Hsinchu, Taiwan, in 2015.

He is currently an Assistant Professor with the Department of Computer Science and Information Engineering, National Central University, Taoyuan, Taiwan. Previously, he was an Assistant Professor with the Department of Computer Science and Engineering, Yuan Ze University, Taoyuan, from February 2018 to January 2020 and a Postdoctoral Fellow with the Institute of Information Science, Academia Sinica, Taipei, Taiwan, from July 2015 to January 2018. His research interests lay in embedded and large-scale storage system designs, emerging memory and storage technologies, and nonvolatile memory systems for machine learning.

Yuan-Hung Kuan received the B.S. degree in computer science and engineering from Tatung University, Taipei, Taiwan, in 2010, and the M.S. degree from the Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, in 2012.

Yuan-Hao Chang (Senior Member, IEEE) received the Ph.D. degree in computer science from the Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, in 2009.

He is currently a Research Fellow with the Institute of Information Science, Academia Sinica, Taipei, where he served as an Associate Research Fellow from March 2015 to June 2018 and an Assistant Research Fellow from August 2011 to March 2015. His research interests include memory/storage systems, operating systems, embedded systems, and real-time systems.

Dr. Chang is a Senior Member of ACM.