IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received September 21, 2020, accepted October 12, 2020, date of publication October 15, 2020, date of current version October 27, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3031322

Level Aware Data Placement Technique for Hybrid NAND Flash Storage of Log-Structured Merge-Tree Based Key-Value Store System

JOONYONG JEONG[1], JAEWOOK KWAK[1], DAEYONG LEE[1], SEUNGDO CHOI[1], JUNGKEOL LEE[1], JUNGWOOK CHOI[1], (Member, IEEE), AND YONG HO SONG[1,2], (Member, IEEE)

[1]Department of Electronics and Computer Engineering, Hanyang University, Seoul 04763, South Korea
[2]Samsung Electronics Company Ltd., Hwaseong 18448, South Korea

Corresponding author: Yong Ho Song (yhsong@hanyang.ac.kr)

This work was supported by the MOTIE/KEIT (Ministry of Trade, Industry and Energy/Korea Evaluation Institute of Industrial Technology) through the Research and Development Program (Developing Processor-Memory-Storage Integrated Architecture for Low-Power, High-Performance Big Data Servers) under Grant 10077609.

ABSTRACT A log-structured merge-tree-based key value store (LSMKV) is an append-only database for storing and retrieving unstructured data, especially in a write-intensive environment. This database uses hierarchical components to store and manage data. Upper-level components have a shorter data lifespan and a higher access locality than lower-level components. Hence, the data access latency of the upper-level components significantly affects the performance of the entire database. Hybrid solid-state drives (SSD) composed of media with different access speeds can improve the performance of an LSMKV by storing the upper-level components using a fast storage space. However, many hybrid SSDs use fast storage spaces to store data that are frequently allocated to the same logical address; they are not suitable for storing append-only component data, which are allocated to adjacent logical addresses. This article proposes a hybrid SSD-management method to reduce the data access latency of append-only LSMKVs and increase the durability of hybrid SSDs. The proposed method allocates the data of upper-level components to a fast storage space using the level information of the data as a hint. This study utilizes dynamic data separation to determine the components to be placed in the fast storage space, NAND block management to store the data with similar lifespans in the same fast NAND block, and a data-relocation method to migrate long-lived data from the fast NAND region to another NAND region. Experimental results indicate that the proposed method reduces the average I/O latency by an average of 12% and increases the device durability by an average of 22%.

INDEX TERMS Data placement, flash storage, hybrid NAND storage, key-value store, log-structured merge tree.

I. INTRODUCTION

Various systems, such as machine learning, blockchain, and content delivery networks, use key-value stores for unstructured data processing. In particular, log-structured merge-tree-based key-value (LSMKV) stores [2], [3], [6], [24] are frequently used in write-intensive application systems, such as financial transactions and social web media. LSMKVs have low write latency as key-value data are written to the storage device in an append-only manner [16].

LSMKVs manage key-value data with an "n-level" log-structured merge-tree (LSM-tree) [19], which is a layered structure of components [2], [6], [15], [23], [29]. The key-value data newly inserted in the storage device are inserted into the top component. The upper component with insufficient space is merged into the lower component. Because of this operational pattern, key-value data of upper-level components, compared to those of lower-level components, are inserted into storage spaces more often [25] and have shorter lifespans [10]. Therefore, the data access latency of the upper-level components significantly affects the performance of the entire LSMKV system.

The associate editor coordinating the review of this manuscript and approving it for publication was Adnan Abid.

Hybrid solid-state drives (SSD) [4], composed of flash memory with different access latencies and storage capacities, can store key-value data of upper-level components in a fast storage space, thereby increasing the speed of an LSMKV while using a large storage space to store the most amount of key-value data. However, many data classification and allocation techniques for hybrid SSDs are not suitable for the access characteristics of an LSMKV. Many previous techniques have classified data that are frequently written to the same logical address as hot data and allocate them to fast storage spaces [1], [7]–[9], [13], [18], [20], [22]. LSMKVs write key-value data to adjacent logical addresses even if the write operation is an update to the key-value data in the storage [10], [16]. Therefore, many existing methods classify most key-value data of LSMKVs as cold data.

This article proposes a data-classification method that verifies the level information of data at the storage device and a data placement method for hybrid SSDs that allocates the data of upper level components to fast storage space, for example, single-level cell (SLC). In a general storage system, the storage device cannot verify the data structure information of the host side. In this study, the level information of the LSM tree data structure can be transferred to the device through an extended host interface.

Two challenges are associated with the proposed data classification and placement method. (1) If the criterion for determining the upper-level components is not adaptively adjusted to the input key pattern, the performance of the proposed method may be degraded. The less duplicated the input key, the less are the data deleted from the input of the merge operation; this leads to an additional merge operation at the lower level. In this case, the storage device writes the key-value data from the top level to the lower level, and the worst case to the bottom level. If the upper level criterion is excessively high, the SLC writes only a small amount of data, which increases the SLC idle time and degrades storage performance. Conversely, if the input key is duplicated frequently, the data merge operation may be terminated at the upper level. In this case, the storage device intensively writes the key-value data of the upper components. If the upper level criterion is extremely low, then the SLC blocks become significantly worn out, and the garbage collection (GC) overhead of the SLC region is increased, thereby deteriorating storage performance. (2) Because the page copy overhead of GC can be increased when data with different lifespans are stored in the same block [27], storing data of various components to the NAND block together can decrease the performance of the proposed method. In a hybrid SSD, GC occurs frequently in an SLC region composed of a small number of SLC NANDs; therefore, page copying the SLC GC degrades the performance of the proposed method. The page copy of the SLC GC can be reduced by storing the data of one component to an SLC block. However, if the upper level criterion is changeable, then some SLC blocks can be retained in the open state for the level that might not be assigned to the SLC region, thereby reducing the available storage space.

To solve the challenges of the proposed method, the following are proposed:

- Data placement method for allocating the data of upper level components to SLC region of hybrid SSD;
- Block management method for placing data of the adjacent level components in the same SLC block;
- Dynamic data separation technique that monitors the amount of data recorded in each NAND region of hybrid SSDs, detects the intensive write to a NAND region as a change in the incoming key pattern of LSMKV workload, and adjusts the write amount in proportion to the endurance of each NAND region;
- Data relocation technique that migrates data of low-level components recorded in the SLC region to another NAND region in accordance with the adjusted data classification criteria of dynamic data separation.

The proposed method was evaluated using an in-house hybrid SSD simulator and storage I/O workload collected from an actual LSM tree-based key-value application. Experimental results show that the proposed method can reduce the average I/O latency by an average of 12.11% and increase the durability of the device by an average of 22.39% compared to the previous method, which uses the request size and access-frequency-based hot/cold separation [9].

The remainder of this article is organized as follows. Chapter 2 provides background information regarding hybrid SSDs and LSMKVs. Chapter 3 provides the storage access and data life characteristics of LSMKVs, which are the goals of this study, and Chapter 4 reviews prior studies related to the storage management of hybrid SSDs. Chapter 5 describes the proposed method, that is, hybrid SSD management in the LSMKV system. Subsequently, the experimental environment and results are presented, followed by the conclusions of this study.

II. BACKGROUND KNOWLEDGE

A. HYBRID SSD

Chang proposed a hybrid SSD that combines an SLC NAND with a fast response time and high cell endurance with an MLC (multi-level cell) NAND with high cell density and low production cost [4]. In the hybrid SSD, the higher the proportion of the SLC block in the entire NAND block, the higher the lifespan and response speed of the flash storage, but the lower the total capacity (Table 1). The recent hybrid SSD can be composed of not only SLC and MLC, but also TLC (triple-level cell) or QLC (quadruple-level cell). Fig. 1 shows the structure of a hybrid SSD composed of SLC and TLC.

According to Chang [4], a hybrid SSD can reduce the storage access latency by placing small and frequently updated data on SLCs. In addition, cold data recorded in SLCs can be relocated to other NAND regions to secure SLC space, and hot data recorded in MLC can be moved to SLCs to reduce data access latency. Meanwhile, to increase the durability of the device, the hybrid SSD should prevent the NAND region from wearing out significantly.

TABLE 1. Comparisons of various level cell flash chips.

Attributes	SLC	MLC	TLC
Read time	25 µs	50 µs	250 µs
Program time	200 µs	1,500 µs	2,700 µs
Maximum P/E cycle	100,000	10,000	2,500
Bit per cell	1 bit	2 bits	3 bits
Cost	Expensive	Middle	Cheap

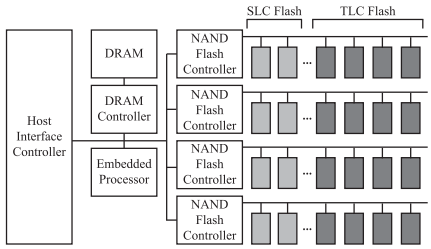


FIGURE 1. Overall architecture of SLC/TLC hybrid SSD.

B. APPEND-ONLY KEY-VALUE APPLICATION: LSMKV

An LSMKV is a "not only structured query language framework" based on a log-structured merge-tree (LSM-tree) [2], [3], [6], [24]. In an LSMKV, the input key-value pair is buffered in the memory and dumped into the storage when accumulated over a certain size. Buffered key-value data in memory is called MemTable. The key-value pair file stored in storage is called a sorted string table (SSTable), which is an immutable file format.

An SSTable is managed in a hierarchical structure called a log-structured merge-tree. The new SSTable is assigned to level 0, which is the highest level. Each level has a limit size, and the lower level has a larger limit size than the upper level. At a level beyond the limit size, compaction is performed. The size limit of the level is not strictly enforced, but functioned as a trigger for compaction.

Because an SSTable is immutable, LSMKVs follow a compaction process to merge and delete existing SStables. Compaction is the data merge operation between levels. At the level where compaction is triggered, an SSTable is selected as the compaction-input. In addition, in the neighboring sublevel of the level at which compaction is triggered, SSTables overlapping the key range of the compaction-input SSTable are selected simultaneously as the compaction-input. Compaction-inputs are read from storage, merged and sorted, and output as new SSTables. The compaction-output SSTables are assigned to the lower level among the two levels where compaction-inputs are collected. On the storage side, the compaction-input SSTables are deleted. As a result of compaction, SSTables at each level are arranged such that the key ranges do not overlap with each other. However, the key range of SSTables at level 0 can be overlapped. Fig. 2 shows the brief architecture of the LSMKV.

III. MOTIVATIONS

A. STORAGE ACCESS PATTERN OF LSM-TREE-BASED KEY-VALUE STORE

Unlike many index structures that write the input data to the same logical address, updating the data in the storage,

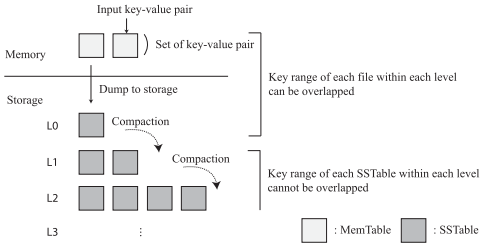


FIGURE 2. Overall architecture of LSM-tree based key-value store.

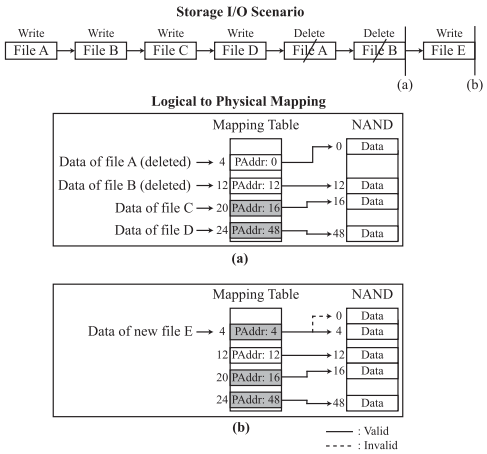


FIGURE 3. Storage workload of a log-structured merge-tree based key-value (LSMKV) store in a given I/O scenario; (a) and (b) present the logical-to-physical mapping status at the pointed time. Each file is an SSTable of LSMKVs. (a) illustrates the allocation of a logical address to a physical address. (b) presents the invalidation and reallocation of a logical address to a physical address. LtoP allocation is invalidated when the LBA of a deleted file is assigned to a new file.

LSMKVs write all the input key-value data to adjacent logical addresses of the storage device in an append-only manner. Fig. 3(a) shows the logical-to-physical address mapping of flash storage when SSTables are inserted and deleted. If new data is assigned to a logical address that has already been assigned to a physical address, the data of an existing physical address are invalidated, and new data are allocated to another physical address (Fig. 3(b)). The operation shown in Fig. 3(b) is similar to the update operation of the flash storage (Fig. 6(a)). However, because SSTables are immutable in LSMKVs, this operation is performed when (1) new data are inserted after the existing SSTable is deleted, or (2) the metadata of the LSMKVs are updated.

B. DATA LIFESPAN CHARACTERISTICS OF LSM-TREE BASED KEY-VALUE STORE

In LSMKVs, one SSTable is assigned to one level, and each level has a capacity limit to store SSTables. In general, the limit size of level $N+1$ is 10 times that of level N. When the level exceeds the limit size, compaction is triggered. SSTables that are selected as compaction inputs are deleted from storage and new SSTables, which are obtained from the compaction, are recorded to storage. Because compaction occurs frequently at a level with a small capacity limit, SSTables at a level with a small limit are likely to be deleted

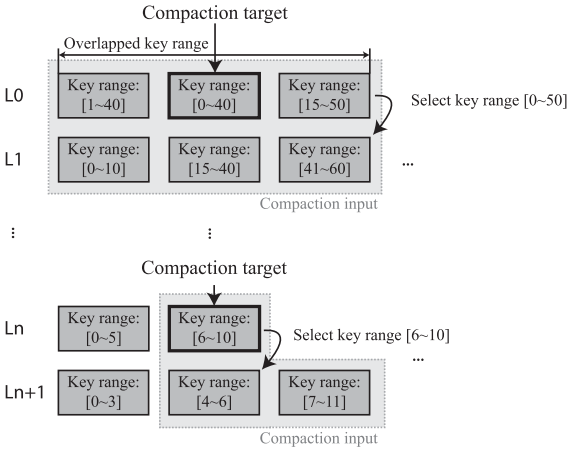


FIGURE 4. Size difference between compaction triggered at levels 0 and n.

from storage owing to compaction. Therefore, upper-level data have a shorter data lifespan and higher access frequency than the lower levels [12]. Herein, the data lifespan refers to the time when a file is created on the host side and deleted from the host.

In addition, owing to the structural characteristics of level 0 SSTables, compaction can occur frequently at levels 0 and 1. All SSTables are constructed through compaction. Compaction is categorized into minor compaction, which is for dumping data at the memory to storage, and major compaction, which is for merging between SSTables stored in storage. Major compaction sorts SSTables such that the key ranges do not overlap with each other to reduce the key seek time. Meanwhile, minor compaction dumps the data recorded in memory into storage regardless of whether the key range is duplicated for the quick flushing of the memory space. Level 0 SSTables, generated as a result of minor compaction, can be selected together as a compaction input because their key ranges overlap. Therefore, compaction triggered at level 0 can be performed with many input SSTables (Fig. 4). Compaction with massive SSTables can result in an overflow of the next level and trigger the next compaction in a chain. Consequently, SSTables from level 0 to 2 are likely to be deleted in the near future compared with the other levels. It is noteworthy that compaction triggered at one level deletes SSTables at the triggered level and at the next level.

Fig. 5 details the average lifespan of an SSTable at each level, where the input key patterns are fully random, zipfian random and latest random (Full-Random-Write, Zipf-Random-Write and Latest-Random-Write of Table 3). Each column value was normalized based on the average lifespan of level 0 SSTables. SSTables of upper-level components, especially levels 0, 1, and 2, have a shorter data lifespan compared to the lower levels. It is noteworthy that the exact lifespan of SSTables may vary depending on the input key pattern because the key range of each SSTable is dependent on the input key pattern. Nevertheless, according to our observations, SSTables of the upper-level had shorter average lifespan than SSTables of the lower-level.

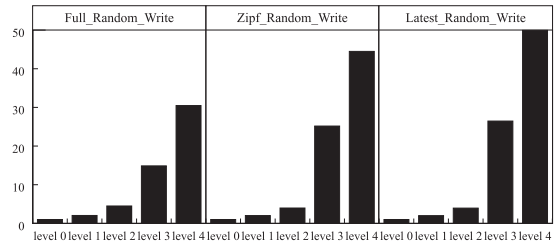


FIGURE 5. Average SSTable lifespan of each level in full random workload, zipfian random workload and latest random workload. Each column value was normalized based on average lifespan of level 0 SSTables.

IV. RELATED STUDIES

To improve the performance of hybrid SSDs, many existing methods allocate the hot data that are frequently written to the same logical address to SLC regions, or use SLCs as a write buffer. In previous methods, data temperature was mainly estimated by the size, sequentiality, or update history of the requested data.

Park *et al.* [20] sent write requests to the SLC region. When the write process is performed a certain number of times, hot/warm data, which are frequently updated, remain in the SLC region, and cold data are relocated to the MLC region.

Im and Shin [8] used SLC areas as a log buffer and MLC areas as a data block. They migrated only cold data that were not frequently updated from SLCs to MLCs but did not migrate data when a large number of page copies was involved in data migration. Furthermore, they assumed large sequential data to write once to cold data, thereby bypassing SLCs and writing it to MLCs.

Lee *et al.* [13] proposed FlexFS, which classifies flash storage devices into SLC and MLC regions. FlexFS predicts the idle time of the workload and suspends the background data migration process when the idle time is insufficient to perform migration between regions. In addition, when the update locality of data recorded in the SLC is high, FlexFS dynamically increases and decreases the sizes of the SLC and MLC regions, respectively.

Im and Shin [9] proposed ComboFTL for an SLC/MLC hybrid NAND flash medium, which included request-size-based data separation and an update-frequency-based data relocation algorithm. In ComboFTL, small random writes are considered as hot data and assigned to SLCs, whereas large sequential writes are considered as cold data and assigned to MLCs. Data recorded in the SLC region are divided into hot and warm data. Hot data are newly recorded data in the SLC region and are classified as warm data if no update occurs during the wrap-around time of the clock. If the warm data are not updated for more than a certain number of GCs, the warm data are migrated to the MLC. Among the data recorded in the MLC, data detected as hot data are migrated to the SLC.

Hong and Shin [7] used an SLC/MLC heterogeneous NAND flash device as a cache for HDDs. They used the SLC and MLC regions as the first- and second-level buffers, respectively; and perform SLC-to-SLC or SLC-to-MLC data migration based on the data update frequency.

However, previously proposed methods for hybrid SSDs are not suitable for the storage access pattern of LSMKVs. First, the data-classification method based on the request size or access sequentiality of data classifies small random and large sequential write data as hot and cold data, respectively (Fig. 6.(b)). Meanwhile, the LSMKV writes key-value data to a storage device as SSTable files, which involves writing large sequential data. Hence, hot/cold classification methods based on the request size or access sequentiality of data are unsuitable for indicating the difference in lifespans between key-value data as they classify most key-value data as cold data. Second, the update-frequency-based data-classification technique estimates the data lifespan using the update frequency of the logical address (Fig. 6.(a)). Because LSMKVs do not check whether a key-value write request is an update request for key-value data in the storage, key-value data are allocated to the logical address space as illustrated in Fig. 3. New SSTables are inserted into adjacent logical addresses, and existing SSTables are deleted. Therefore, update-frequency-based data classification is inadequate for estimating the access likelihood of an SSTable.

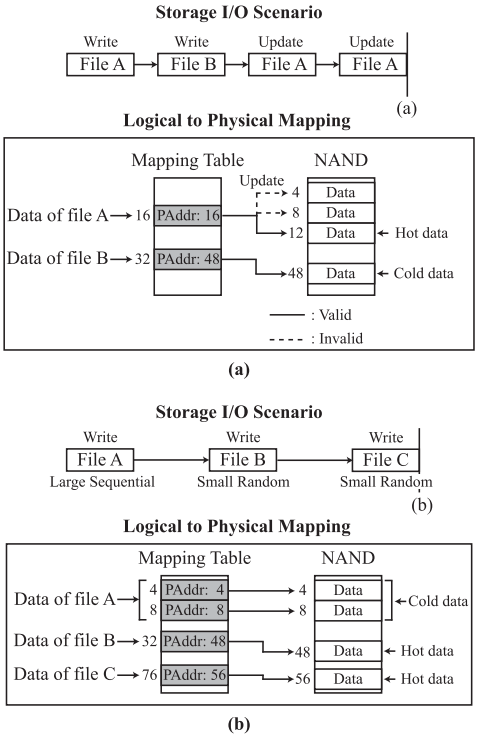


FIGURE 6. Basic concepts of traditional hot/cold-based data-separation: (a) update-frequency-based data-separation and (b) request-size- & access-sequentiality-based data-separation.

V. MAIN IDEA

A. DATA PLACEMENT TECHNIQUE OF HYBRID SSD FOR LOG-STRUCTURED MERGE-TREE BASED KEY-VALUE STORE

This study aims to effectively estimate the lifespan of SSTables in storage and allocate short-lifespan SSTables to the SLC NAND region. The proposed method verifies the level of each SSTable in the storage and allocates upper-level SSTables to the SLC region. Because the file-deletion time at the host does not coincide with the data-invalidation time at the storage, the proposed method uses the TRIM command of the SSD to notify the logical block address (LBA), which is no longer used by the device when the file is deleted from the host. In addition, to enable the storage to obtain the level information of each SSTable, the proposed technique uses an extended host interface that can include the level information of each SSTable. Fig. 7 shows the overall architecture of the SLC/TLC hybrid SSD system using the proposed method.

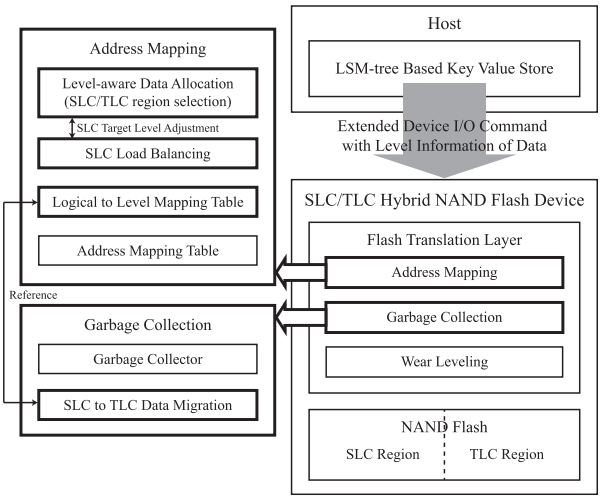


FIGURE 7. Overall architecture of the proposed scheme. The highlighted box represents the scope of this study.

The proposed data-placement method is subsequently detailed. The host sends a data-write command containing the level information of the data to the storage. To set the initial value of the level depth to be assigned to the SLC, the storage initially records all the data to the SLC and monitors the level depth of the incoming data. The level depth to be included in the SLC is denoted as *SLCTargetLevel* herein. When the first GC of the SLC region is performed, the storage sets the lowest level recorded in the SLC as *SLCTargetLevel*. For subsequent data-write requests, the storage allocates data whose level is higher than *SLCTargetLevel* to the SLC. When data is written to the SLC, the storage media updates the logical-to-level mapping table (LtoLV table) that maps the data level and the logical address of the SLC page. The LtoLv table is used to select the page to migrate from the SLC region to the TLC region. Data migration is described at the end of this section. The LtoLv table does not replace the address-mapping table managed by the existing flash storage but complements it. When all the blocks in the SLC region are filled with valid pages, incoming data are allocated to the TLC even if their levels are higher than *SLCTargetLevel*. Algorithm 1 shows the proposed data-allocation technique after the initial *SLCTargetLevel* is determined.

Algorithm 1 Data Allocation Process With Level Information

Input: page address, data size, data, level information of data (LevelInfo)

Output: none

```
1: if LevelInfo ≤ SLCTargetLevel then
2:     if writable space of SLC > data size then
3:         update the address mapping table;
4:         update the logical to level mapping table;
5:         OpenBlock_SLC ← GetSLCOpen-
   Block(LevelInfo);
6:         write data to OpenBlock_SLC.freePage;
7:         LoadBalancingManager(WRITE, DataSize);
8:         SLCLoadBalancer(WRITE);
9:     else
10:        update the address mapping table;
11:        write data to WritableBlock_TLC.freePage;
12:        WriteAmount_TLC + = DataSize;
13:        LoadBalancingManager(OVERFLOW, 0);
14:        SLCLoadBalancer(OVERFLOW);
15:    end if
16: else
17:    update the address mapping table;
18:    write data to WritableBlock_TLC.freePage;
19:    WriteAmount_TLC + = DataSize;
20: end if
```

B. DYNAMIC DATA SEPARATION AND LOAD BALANCING OF HYBRID SSD FOR LOG-STRUCTURED MERGE-TREE-BASED KEY-VALUE STORE

SLCTargetLevel, the data placement criterion of the proposed method, is adjusted in two cases. First, the proposed method reduces the *SLCTargetLevel* when the SLC GC cannot obtain a free block because the SLC region is filled with valid pages (SLC overflow). This method can reduce the average lifespan of data to be included in the SLC region. Valid data included in the SLC are removed from the SLC region through data deletion from the host or data migration. Adjustment of *SLCTargetLevel* due to SLC overflow is performed once whenever data are accumulated in the SLC region by the size of the SLC region.

Second, *SLCTargetLevel* is adjusted when intensive writes to any NAND region of the hybrid SSD occurs. The proposed method monitors the amount of data input to each NAND region of the hybrid SSD. Different types of NAND have different maximum program/erase (P/E) cycles; therefore, the proposed method adjusts the ratio of the amount of input data between NAND regions to be similar to the durability ratio between NAND regions. Herein, the *Durability* of the SLC and TLC regions is the total erase count until all blocks in each region are worn out, expressed as the product of the number of blocks in the region and the maximum P/E cycle per block($Durability = Block\_Count \times Max\_PE\_Cycles\_per\_Block$).

Because the data write of flash storage is performed in a page unit, the total write amount of a NAND region before a wear-out is $LifelongWriteAmount = Pages\_per\_Block \times Durability$. Finally, the write amount ratio of the SLC region compared with that of the TLC region proposed herein, $WriteAmountBalance(\phi)$, is expressed as follows:

$$\begin{aligned}\phi &= \frac{LifelongWriteAmount\_SLC}{LifelongWriteAmount\_TLC} \\ &= \frac{Pages\_per\_Block\_SLC}{Pages\_per\_Block\_TLC} \\ &\quad \times \frac{Block\_Count\_SLC}{Block\_Count\_TLC} \\ &\quad \times \frac{Max\_PE\_Cycles\_per\_Blk\_SLC}{Max\_PE\_Cycles\_per\_Blk\_TLC}\end{aligned}$$

The proposed method adjusts the write amount of the SLC region by increasing or decreasing the *SLCTargetLevel* such that the ratio of the amount of input data between NAND regions and the durability ratio between NAND regions become similar. Algorithm 3 shows how the adjustment must be made. The appropriate write amount for the SLC region proposed herein was derived by multiplying the amount of data written in the TLC region by $WriteAmountBalance(\phi)$. The proposed method increases and decreases $WriteAmountBalance(\phi)$ to a constant *BalancingWeight* to determine the lower and upper bounds for the write volumes of the SLC region (lines 3 and 4 of Algorithm 3). Consequently, the lower limit for the write amount of the SLC region is calculated as $(\phi - BalancingWeight) \times WriteAmount\_TLC$ (line 5 of Algorithm 3), and the upper limit write amount is calculated as $(\phi + BalancingWeight) \times WriteAmount\_TLC$ (line 7 of Algorithm 3). If the amount of data recorded in the SLC is less than the lower limit of the write amount, then *SLCTargetLevel* is increased by 1 (lines 5 and 6 of Algorithm 3). Conversely, if the amount of data recorded in the SLC exceeds the upper limit of the write amount, then *SLCTargetLevel* is decreased by 1 (line 7 and 8 of algorithm 3). Algorithm 2 represents a sequence that determines when *SLCTargetLevel* is adjusted. Adjusting the write overhead between NAND regions is performed once every time data are accumulated in the SLC region (line 3 of Algorithm 3), as in the SLC overflow. However, the execution cycle is independent of the SLC overflow (line 14 of Algorithm 3).

C. SLC BLOCK MANAGEMENT WITH DYNAMIC DATA SEPARATION

The proposed method allocates the data of the same or adjacent level to the same SLC block to reduce the page copy overhead that occurs in the GC of the SLC region. Suppose that level N data are requested to be written to the device when *SLCTargetLevel* is larger than N. The device selects a writable block from the free block pool of the SLC region and sets it as a "level N open block" for recording level N data. If no writable page occurs in a level N open block, then the

Algorithm 2 LoadBalancingManager: Calculate Data Write Amount of SLC Region, to Adjust the *SLCTargetLevel*

Input: source variable (Source), data size
Output: none

```
1: if Source == WRITE then
2:     WriteAmount_SLC + = DataSize;
3:     if OverflowFlag == TRUE then
         //to adjust the load balancing cycle after SLC overflow occurs
4:         WriteAmount_SLC_Overflow + = DataSize;
5:     end if
6: else if Source == OVERFLOW then
7:     OverflowFlag ← TRUE;
8: end if
```

Algorithm 3 SLCLoadBalancer: Adjust the *SLCTargetLevel*

Input: source variable (Source)
Output: none

```
1: if Source == WRITE then
2:     if WriteAmount_SLC ≥ RegionSize_SLC then
3:         LoadBalancer_LowerBound ← (WriteAmount-
   Balance − BalancingWeight);
4:         LoadBalancer_UpperBound ← (WriteAmount-
   Balance + BalancingWeight);
5:         if WriteAmount_SLC < LoadBalancer_LowerBound*WriteAmount_TLC then
6:             SLCTargetLevel++;
7:         else if WriteAmount_SLC > LoadBalancer_UpperBound*WriteAmount_TLC then
8:             SLCTargetLevel−−;
9:         end if
10:        WriteAmount_SLC ← 0;
11:        WriteAmount_TLC ← 0;
12:    end if
13: else if Source == OVERFLOW then
14:    if (OverflowFlag == TRUE) & (WriteAmount_SLC_Overflow ≥ RegionSize_SLC) then
15:        SLCTargetLevel−−;
16:        WriteAmount_SLC_Overflow ← 0;
17:        OverflowFlag ← FALSE;
18:    end if
19: end if
```

level N block is closed. If *SLCTargetLevel* becomes smaller than N when the level N block is open, then level N data are no longer allocated to the SLC region; therefore, the level N block may remain open. In this case, the device expires the level N open block. The expired level N block is then selected as the top priority when selecting an open block for recording level $N-1$ data and becomes a level $N-1$ open block. If *SLCTargetLevel* becomes N again before the expired level N block becomes a level $N-1$ open block and level N data

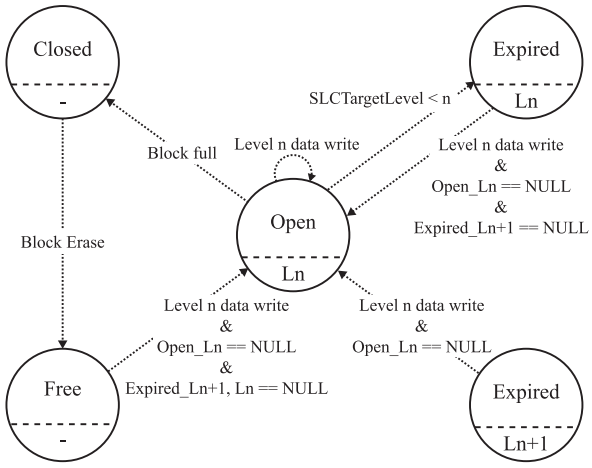


FIGURE 8. State diagram of SLC block management for LSMKV. The statement in the circle on the dotted line represents the block state, and that under the dotted line the block level.

starts to be allocated to the SLC, the expired level N block is reopened for level N data. Fig. 8 shows the state diagram of the SLC block management method proposed herein.

D. SLC TO TLC DATA MIGRATION WITH DYNAMIC DATA SEPARATION

The long-lived data included in the SLC region can reduce the available space of the SLC, causing frequent SLC GC or increasing the valid page copy overhead during GC. Hence, when GC is performed in the SLC region, the proposed method examines the data of lower-level components recorded in the SLC and relocates the data to the TLC. To track the level of data recorded in the SLC, the proposed method utilizes the LtoLv table, which links the logical address of the SLC page with the level information of the data allocated to it (line 4 of Algorithm 1). To determine the SLC block to be relocated, the device verifies the average data level of pages in the SLC blocks when GC is performed in the SLC region (lines 3 and 4 of Algorithm 4). The device selects the block with the highest average level as a relocation candidate, called *MigrationCandidateBlock* (line 5 of Algorithm 4). If the average level of pages in the *MigrationCandidateBlock* exceeds *SLCTargetLevel*, then the candidate is set to *MigrationVictimBlock* (lines 8 and 9 of Algorithm 4). In *MigrationVictimBlock*, every valid-page whose level exceeds *SLCTargetLevel* is relocated to the TLC(line 7 to 10 of Algorithm 5). Algorithm 4 shows the data-block-selection method for SLC-to-TLC relocation, and Algorithm 5 shows the method for SLC-to-TLC data migration.

The proposed method does not migrate data from the TLC to the SLC; therefore, the LtoLv table does not store the level information of data included in the TLC. Key-value data below *SLCTargetLevel* will be invalidated owing to the compaction in the near future, and future input of the key-value data below *SLCTargetLevel* are allocated to the SLC region. Therefore, TLC key-value data whose level is

Algorithm 4 GetMigrationVictimfromSLC: Select the Migration-Victim-Block

Input: none
Output: block number of the MigrationVictimBlock

```
MigrationVictimBlock ← NULL;
if SLC region needs the garbage collection then
    for each block of SLC region do
        check average level of pages in block;
        choose the block, which has highest average level of pages, to MigrationCandidateBlock;
    end for
end if
if average level of pages in MigrationCandidateBlock > SLCTargetLevel then
    MigrationVictimBlock ← MigrationCandidateBlock;
    return MigrationVictimBlock;
else
    return NULL;
end if
```

Algorithm 5 SLC to TLC Data Migration

Input: none
Output: none

```
MigrationVictimBlock = GetMigrationVictimfromSLC();
if MigrationVictimBlock != NULL then
    for each page of MigrationVictimBlock do
        Get logical address of the page;
        if the page has valid logical address then
            Level ← GetLevelInformation(page);
            if (Level > SLCTargetLevel) then
                update the address mapping table;
                invalidate the level mapping of page;
                migrate the page from SLC to TLC;
            end if
        end if
    end for
end if
```

less than *SLCTargetLevel* do not require placement correction. In addition, the LtoLv table for the TLC region is required for TLC-to-SLC data migration. A massive LtoLV table, including the level information of the TLC area, has a longer search time and a larger storage cost.

VI. EVALUATION

A. EXPERIMENTAL SETTINGS

To evaluate the proposed method, we executed a benchmark of the LSMKV application to extract the storage workload and then executed the workload in a storage simulator. The storage simulator was an in-house event-driven simulator for the SLC/TLC hybrid SSD. Our simulator is implemented based on Flashsim [11], a widely used event-driven SSD simulator. Event-driven SSD simulators wrap the I/O request event from the host as an FTL event. The FTL event is processed in units of NAND pages when reading and writing and in units of NAND blocks when erasing. The unit size of the event-processing can be more than the size of a single NAND page or a single NAND block, if the internal parallelism of SSD is exploited. By modifying Flashsim, the simulator of this study includes SLC/TLC hybrid SSD hardware components, the proposed FTL, and ComboFTL. Table 2 shows the SSD structure and NAND cell specifications used in the experiment.

The LSMKV application used in the experiment was RocksDB [2]. We generated the key-value I/O for RocksDB using DBBench, a built-in benchmark tool of the RocksDB framework, and YCSB [5], one of the widely used benchmark tool for cloud systems and databases. We extracted the storage I/O commands which are requested from RocksDB, including the data level information. The three key input patterns used in the experiment, as summarized in Table 3, were the full-random key order, which is the basic scenario of DBBench; the zipfian distribution random order (zipf), which is frequently used for the performance evaluation of key-value stores [14], [17], [21], [26], [28]; and the latest distribution random order (latest), one of the distribution options of YCSB. Zipf- and latest-random workloads are skewed random workloads in which specific keys are frequently inserted. The compaction operation of LSMKV deletes more duplicate keys in zipf- and latest-random workloads (especially in the latest-random workload) than in full-random workload because key duplication occurs more frequently in the zipf- and latest-random workloads than in the full-random workload. Therefore, in the zipf- and latest-random workload, the average size of the compaction output is smaller than that in a full-random workload. Furthermore, the compaction does not cascade down to the lower level but ends at the upper level.

For the experiment, we performed a comparison among the ratios of SLC blocks in the total NAND block of 1%, 2%, 3%, 5%, and 10% (see Table 4). The overprovision block ratio (that is, the additional space excluding the user area among the physical areas) of the SSD used in the experimental device was 20%.

In the experiment, the proposed method was compared with the existing request-size- & LBA-update-frequency-based hot/cold separation method, ComboFTL [9]. ComboFTL was originally proposed for SLC/MLC hybrid SSDs; however, in this study, all experiments were performed for SLC/TLC hybrid SSDs. The proposed method uses the greedy algorithm as the GC victim selection policy. The greedy algorithm selects the block with the highest invalid page count as the victim block of the GC. In this experiment, we set *BalancingWeight* of the proposed SLC load-balancing technique to 0.1.

B. EXPERIMENTAL RESULTS

Fig. 9 shows a comparison of the performances of ComboFTL and the method proposed herein. The x- and y-axes

TABLE 2. Configurations of the simulated SSD.

	Specification		Structure		
SLC Mode	Page Read	50 μs	Page Size		16 KB
	Page Program	300 μs	Channel Count		4
	Block Erase	10 ms	Way(chip) per Channel		2
	Max. P/E Cycle	100,000	Plane per Way(chip)		1
TLC	Page Read	250 μs	Pages per Block	SLC Mode	256
	Page Program	2500 μs		TLC	768
	Block Erase	10 ms	GC Free Block Retention Ratio	SLC Region	10%
	Max. P/E Cycle	2,500		TLC Region	10%

TABLE 3. Workload setting.

	Tool	Read/Write Ratio	Record Counts	Description
Full_Random_Write	DBBench	0:100	1K × 200,000,000	Keys are requested in a random uniform pattern
Zipf_Random_Write	YCSB	0:100	1K × 200,000,000	Keys are randomly requested, according to the zipfian
Zipf_Random_UpdateHeavy	YCSB	50:50	1K × 100,000,000	distribution that some keys are frequently requested
Latest_Random_Write	YCSB	0:100	1K × 200,000,000	Most recent keys are most frequently requested
Latest_Random_UpdateHeavy	YCSB	50:50	1K × 100,000,000	

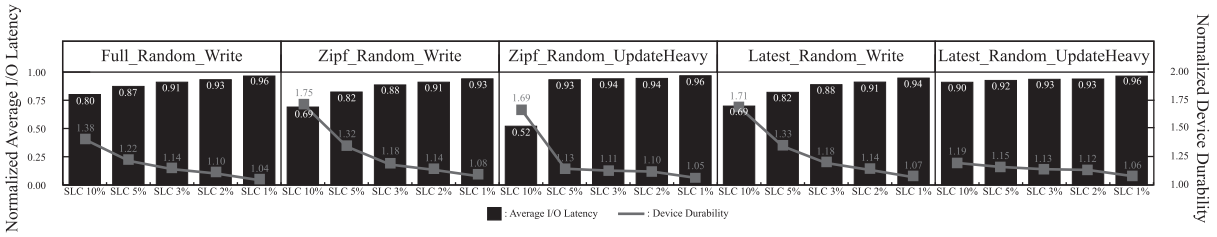


FIGURE 9. Average I/O latency and durability of the hybrid SSD of the proposed method, compared to the ComboFTL. Boxes represent the latency (lower is better), and lines show the device durability (higher is better). Each value was normalized to the result of ComboFTL for the given condition.

TABLE 4. Simulator storage space setting.

	Proportion of SLC Region				
	10%	5%	3%	2%	1%
# of TLC Blocks	29,492	31,130	31,785	32,113	32,441
# of SLC Blocks	3276	1638	983	655	327
TLC Region Size	345 GB	364 GB	372 GB	376 GB	380 GB
SLC Region Size	12.8 GB	6.4 GB	3.8 GB	2.5 GB	1.3 GB

represent the proportions of SLC blocks in the total NAND blocks and performance measurements, respectively. Each column of the column graph represents the normalized average I/O latency, which corresponds to the left y-axis value. The line graph shows the normalized durability of the hybrid SSD, which corresponds to the right y-axis. All values in Fig. 9 were normalized to the measurement results of ComboFTL under the corresponding experimental conditions. For example, in a full-random-write workload, when the proportion of the SLC region was 10%, the average I/O latency of the proposed method was 0.80 times, and device durability was 1.38 times that of the ComboFTL. In the experiment performed in this study, the device durability was measured by the number of workload iterations until all NAND blocks in any NAND region were worn out. Hence, in the experiment, even if one NAND region was still writable, the hybrid SSD was considered to be unusable if all NAND blocks of the other NAND region were worn out. The durability of the hybrid SSD was calculated as $MIN(\frac{Durability\_SLC}{EraseCount\_SLC}, \frac{Durability\_TLC}{EraseCount\_TLC})$. *EraseCount_NAND* is the total erase count of the NAND region when the workload is performed once. When the *EraseCount_NAND* was 0, the value of $\frac{Durability\_NAND}{EraseCount\_NAND}$ was replaced with ∞ in our experiment. As a reminder, the *Durability* of the SLC and TLC regions was derived by calculating *Block_Count* × *Max_PE_Cycles_per_Block* in this study.

1) RESULTS: AVERAGE I/O LATENCY

As shown in Fig. 9, the proposed method has a shorter average I/O latency than ComboFTL and reduces the average I/O latency more as the SLC region size increases. In the LSMKV environment, the size-based data separation of ComboFTL records only a small amount of data (only the small-random writes) to the SLC. Meanwhile, as the size of the SLC region increases, the proposed method stores more data (SSTables) to the SLC, by increasing *SLCTargetLevel*.

Fig. 10 also shows that the proposed method writes a larger amount of data to the SLC as the SLC region size increases. The x-axis represents the applied schemes; Base is ComboFTL, and Prop is the proposed method. The y-axis represents the ratio of data I/O recorded in SLC and TLC out of total data I/O. The column value in Fig. 10 (a) was normalized to the total amount of data read from storage by the host for each workload, and the column value in Fig. 10 (b) was normalized to the total amount of data written from the host to storage for each workload. For example, in the full-random-write workload of (b), when the SLC region ratio was

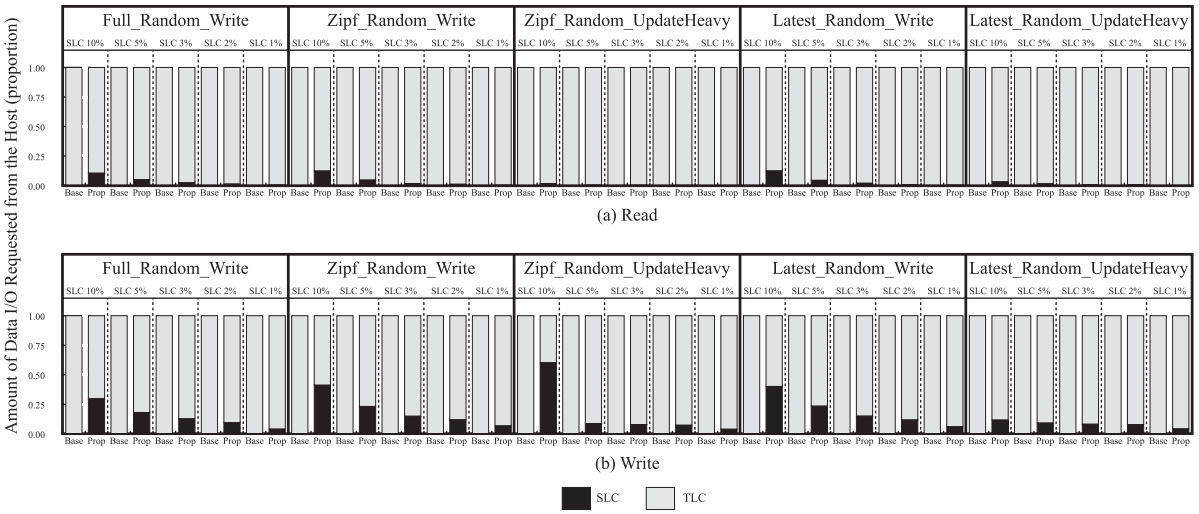


FIGURE 10. Proportion of the data I/O request arriving at the SLC and TLC. The column values were normalized to the total read (a) or write (b) data size of each workload, which is requested from the host to the storage. The bold line of the box separates the experimental results by the case of each workload.

10%, the proposed method allocates 29.66% of the total data written to storage by the host to the SLC and allocates 3.91% of the total write-requested data to the SLC when the SLC region ratio was 1%.

Exceptionally, in the latest-random-update-heavy workload, when the SLC region ratio was 10%, the proposed method wrote a smaller amount of data to the SLC than the other workloads (as shown in Fig. 10 (b)). Hence, the average I/O latency reduction was small due to the characteristics of the input key pattern of the latest-random-workload. The latest-random-workload exhibits strong temporal locality of the key-value operations [25], and thus frequently updated 'latest' key-value pairs exist in the upper-level SSTable. As a result, many compactions occurring in the latest-random-workload did not cascade down to a lower level, but ended at an upper level. Therefore, the *SLCTargetLevel* of the proposed method in the latest-random workload remained smaller than in the case of other workloads. In latest-random-update-heavy and zipf-random-update-heavy workloads, L1 (level 1) was the highest in terms of the amount of data written to storage for each level of LSMKV. In the latest-random-update-heavy, storage writes of L1 SSTables were about 61% of the total storage writes of SSTables; 10% in full-random-write, 18% in zipf-random-write and latest-random-write, and 53% in zipf-random-update-heavy. Therefore, the data write size to the SLC region significantly changed depending on whether L1 SSTables were allocated to the SLC. In the latest-random-update-heavy workload, when the SLC region ratio was 10%, the proposed method had difficulties in determining the appropriate *SLCTargetLevel* that enabled the SLC and TLC to wear-out similarly.

Fig. 11 shows the classification of the NAND I/O operation. The NAND read/write operation of the proposed method is classified as a page copy for GC, requested page read/write to serve the host-request, and SLC page read/TLC page write for SLC-to-TLC data kickout (migration). Data migration between the SLC and TLC of ComboFTL was considered a copy operation for GC. The x-axis represents the applied schemes, and the y-axis denotes the ratio of data I/O size of the three NAND I/O operations relative to the total NAND I/O size. Each column of Fig. 11 (a) was normalized to the total data read size for the SLC region; Fig. 11 (b) was normalized to the total data write size for the SLC region; and Fig. 11 (c) was normalized to the total data write size for the TLC region in each given environment. As a valid page copy in the TLC GC did not exist, all reads to the TLC region were requested from the host. Note that the total data size, which is the basis for normalization of each column, is different for each column. Even if the workload and SLC region ratio are the same, the amount of data recorded in each NAND region is different if the applied scheme is different. For example, in the zipf-random-update-heavy workload, when the SLC region ratio is 2%, 19.66% of total SLC read operations were page copy operations for GC in ComboFTL, and 21.33% of total SLC reads were page copies for GC in the proposed method. This appears similar, but the size of the copied data for GC is about 30 times greater. This is because the proposed method stored more data in the SLC, than the ComboFTL.

Overall, the SLC GC overhead of the proposed scheme is higher in the write-only workload than in the update-heavy workload. This is because in both cases, the key-value pair with high temporal locality is inserted, but the write-intensity is different between workloads. The more intensively the KV pair write is requested, the more frequent and larger compaction occurs. Generally, because compaction causes storage writes, the more write requests of the key-value pair are intensively requested, the more storage writes occur. Herein, compaction occurs more frequently in the upper-level than in

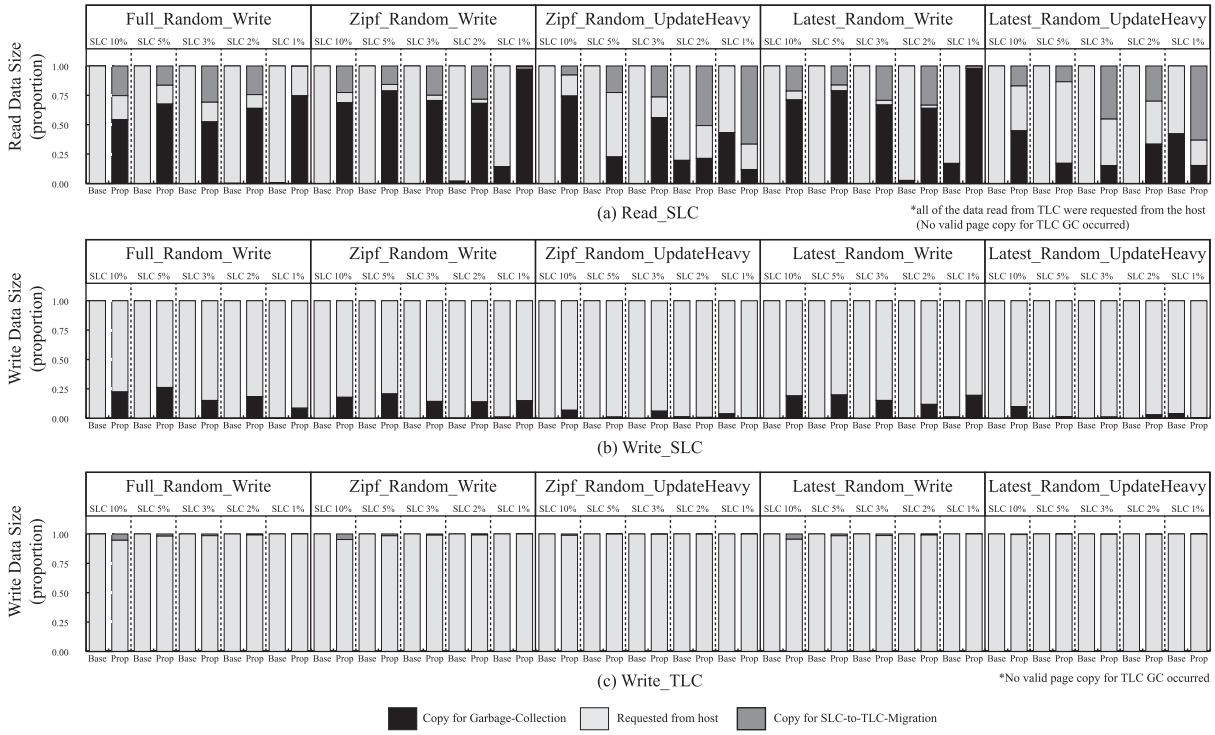


FIGURE 11. NAND I/O operation breakdown. Each column was normalized to the total write or read data size for each corresponding environment: each column in (a) was normalized to the total amount of reads to SLC, (b) was normalized to the total amount of writes to SLC, and (c) was normalized to the total size of writes to TLC. Since all reads to the TLC were requested from the host (no valid page copy read for the GC occurred), the graph for the reads to TLC region was not presented from the figure.

the lower-level, and in general, the size of the SLC region is smaller than the size of the TLC region. Therefore, when the key-value pair write is intensively requested, the SLC GC of the proposed method occurs frequently. If the SLC GC victim block is not fully deleted (e.g., in an environment where compaction is performed in multi-threads, the L2 SSTables are being recorded in the SLC, but the L0 SSTables have not yet been deleted), the GC overhead increases because of the valid page copy overhead. As shown in Fig. 11 (a), if the size of the SLC area is insufficient or the SLC has long-lived data, the copy overhead of SLC GC is high. Nevertheless, because the proposed method allocates a larger number of data writes requested by the host to the SLC than the ComboFTL, the average I/O latency can be reduced.

2) RESULTS: DEVICE DURABILITY

The line graph in Fig. 9 shows the device durability of of the proposed method compared to ComboFTL. The proposed method increases the device durability of hybridSSD in the LSMKV operating environment compared to ComboFTL. Because the amount of data which are requested as small random-writes are not large in the LSMKV operating environment, the amount of host-requested data which were allocated to the SLC region space, in the case of ComboFTL, was from 0.001 to 0.033 multiplied that of the proposed scheme. Owing to the SLC under-utilization of ComboFTL, the SLC NANDs wear less and the TLC NANDs wear more in ComboFTL than in the proposed method. This causes a difference in device durability between ComboFTL and the proposed method.

Generally, the difference in device durability between the two methods increases as the SLC region ratio increases. However, the proposed method did not significantly improve the device durability compared to other workloads in the latest-random-update-heavy workload, when the SLC region ratio was 10%. This is for the same reason that the proposed method did not significantly improve the average I/O latency in the same experimental environment, as mentioned in the section above. When SSTables of a specific level are intensively input to storage (e.g. L1, in case of the zipf and latest-random-update-heavy workload), the proposed method repeatedly excludes and includes the level from *SLCTargetLevel*. If the SSTables of the corresponding level are stored in the SLC, the SLC is worn out excessively; conversely, if not stored in the SLC, then the SLC is underutilized.

VII. CONCLUSION

A hybrid SSD-management technique for an LSMKV system was proposed. The proposed method monitored the level information of the LSMKVs of the device and estimated that the lifespan of the data of higher-level components was shorter than that of the data of lower-level components. This method stored the data of upper-level components in the SLC and the data of lower-level components in the TLC. Data that were included in the same level component were assigned

to the same SLC block. Subsequently, the proposed method adjusted the data placement amount of the SLC region such that the ratio of data written to the SLC and TLC regions was similar to the durability ratio of the two NAND regions. The experiment indicated that the proposed method can reduce the average I/O latency and increase the average device durability by approximately 12.11% and 22.39% compared with the existing method, respectively.

REFERENCES

[1] A. I. Alsalibi, S. Mittal, M. A. Al-Betar, and P. B. Sumari, "A survey of techniques for architecting SLC/MLC/TLC hybrid flash memory-based SSDs," *Concurrency Comput., Pract. Exper.*, vol. 30, no. 13, p. e4420, Jul. 2018.

[2] D. Borthakur, "Rocksdbl a persistent key-value store," Facebook, Inc., Menlo Park, CA, USA, Tech. Rep., 2014. [Online]. Available: http://rocksdb.org/

[3] 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.*, vol. 26, no. 2, pp. 1–26, Jun. 2008.

[4] L.-P. Chang, "Hybrid solid-state disks: Combining heterogeneous NAND flash in large SSDs," in *Proc. Asia South Pacific Design Autom. Conf.*, Jan. 2008, pp. 428–433.

[5] 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. (SoCC)*, 2010, pp. 143–154.

[6] S. Ghemawat and J. Dean, "Leveldb," Google, Inc., Mountain View, CA, USA, Tech. Rep., 2011. [Online]. Available: https://github.com/google/leveldb

[7] S. Hong and D. Shin, "NAND flash-based disk cache using SLC/MLC combined flash memory," in *Proc. Int. Workshop Storage Netw. Archit. Parallel I/Os*, May 2010, pp. 21–30.

[8] S. Im and D. Shin, "Storage architecture and software support for SLC/MLC combined flash memory," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2009, pp. 1664–1669.

[9] S. Im and D. Shin, "ComboFTL: Improving performance and lifespan of MLC flash memory using SLC flash buffer," *J. Syst. Archit.*, vol. 56, no. 12, pp. 641–653, Dec. 2010.

[10] T. Kim, D. Hong, S. S. Hahn, M. Chun, S. Lee, J. Hwang, J. Lee, and J. Kim, "Fully automatic stream management for multi-streamed SSDs using program contexts," in *Proc. 17th USENIX Conf. File Storage Technol.*, 2019, pp. 295–308.

[11] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "FlashSim: A simulator for NAND flash-based solid-state drives," in *Proc. 1st Int. Conf. Adv. Syst. Simulation*, Sep. 2009, pp. 125–131.

[12] J. Lee, W. G. Choi, D. Kim, H. Sung, and S. Park, "TLSM: Tiered log-structured merge-tree utilizing non-volatile memory," *IEEE Access*, vol. 8, pp. 100948–100962, 2020.

[13] S. Lee, K. Ha, K. Zhang, J. Kim, and J. Kim, "Flexfs: A flexible flash file system for MLC NAND flash memory," in *Proc. USENIX Annu. Tech. Conf.*, 2009, pp. 1–14.

[14] H.-S. Lim and J.-S. Kim, "LevelDB-raw: Eliminating file system overhead for optimizing performance of LevelDB engine," in *Proc. 19th Int. Conf. Adv. Commun. Technol. (ICACT)*, 2017, pp. 777–781.

[15] L. Lu, T. S. Pillai, H. Gopalakrishnan, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "WiscKey: Separating keys from values in SSD-conscious storage," *ACM Trans. Storage*, vol. 13, no. 1, pp. 1–28, Mar. 2017.

[16] C. Luo and M. J. Carey, "LSM-based storage techniques: A survey," *VLDB J.*, vol. 29, no. 1, pp. 393–418, Jan. 2020.

[17] P. Menon, T. Rabl, M. Sadoghi, and H.-A. Jacobsen, "CaSSanDra: An SSD boosted key-value store," in *Proc. IEEE 30th Int. Conf. Data Eng.*, Mar. 2014, pp. 1162–1167.

[18] B.-W. Nam, G.-J. Na, and S.-W. Lee, "A hybrid flash memory SSD scheme for enterprise database applications," in *Proc. 12th Int. Asia–Pacific Web Conf.*, Apr. 2010, pp. 39–44.

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

[20] S.-H. Park, J. Park, J. Jeong, J. Kim, and S. Kim, "A mixed flash translation layer structure for SLC-MLC combined flash memory system," in *Proc. 1th Int. Workshop Storage I/O Virtualization, Perform., Energy, Eval. Dependability (SPEED)*, 2008, pp. 1–6.

[21] K. Ren, Q. Zheng, J. Arulraj, and G. Gibson, "SlimDB: A space-efficient key-value storage engine for semi-sorted data," *Proc. VLDB Endowment*, vol. 10, no. 13, pp. 2037–2048, Sep. 2017.

[22] M. Sung and K. Kim, "Asymmetric flash volume management," *IEEE Trans. Consum. Electron.*, vol. 58, no. 2, pp. 455–461, May 2012.

[23] Y. Tang, A. Iyengar, W. Tan, L. Fong, and L. Liu, "Write-optimized indexing for log-structured key-value stores," Georgia Inst. Technol., Atlanta, GA, USA, Tech. Rep. GIT-CERCS-14-01, 2014.

[24] M. Nalin Vora, "Hadoop-HBase for large-scale data," in *Proc. Int. Conf. Comput. Sci. Netw. Technol.*, vol. 1, Dec. 2011, pp. 601–605.

[25] 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 Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 563–568.

[26] 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.*, 2015, pp. 71–82.

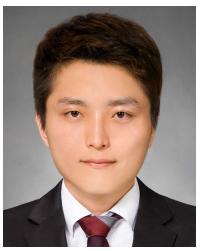
[27] W. Xie, Y. Chen, and P. C. Roth, "ASA-FTL: An adaptive separation aware flash translation layer for solid state drives," *Parallel Comput.*, vol. 61, pp. 3–17, Jan. 2017.

[28] T. Yao, J. Wan, P. Huang, X. He, Q. Gui, F. Wu, and C. Xie, "A light-weight compaction tree to reduce I/O amplification toward efficient key-value stores," in *Proc. 33rd Int. Conf. Massive Storage Syst. Technol. (MSST)*, 2017, pp. 1–13.

[29] J. Zhang, Y. Lu, J. Shu, and X. Qin, "Flashkv: Accelerating kv performance with open-channel SSDs," *ACM Trans. Embedded Comput. Syst. (TECS)*, vol. 16, no. 5s, p. 139, 2017.

JOONYONG JEONG received the B.S. degree from the Department of Information System, Hanyang University, Seoul, South Korea, in 2015. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University.

His research interests include NAND flash-based storage systems, databases, and key-value stores.

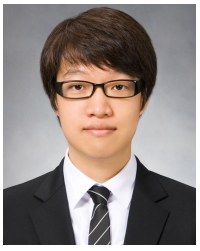


JAEWOOK KWAK received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2015. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University.

His research interests include computer architecture, embedded systems, and NAND flash-based storage systems.

DAEYONG LEE received the B.S. degree from the School of Electronic Engineering, Soongsil University, Seoul, South Korea, in 2014, and the M.S. degree from the Department of Electronics and Computer Engineering, Hanyang University, Seoul, in 2017, where he is currently pursuing the Ph.D. degree.

His research interests include embedded systems and NAND flash memories.



SEUNGDO CHOI received the B.S. and M.S. degrees in electronics and computer engineering from Hanyang University, Seoul, South Korea, in 2012 and 2014, respectively, where he is currently pursuing the Ph.D. degree in electronics and computer engineering.

His research interests include high-performance computing, computer architecture, and low-power systems.

JUNGKEOL LEE received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2014. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University.

His research interests include embedded computing and the IoT device.

JUNGWOOK CHOI (Member, IEEE) received the B.S. and M.S. degrees in electrical and computer engineering from Seoul National University, South Korea, in 2008 and 2010, respectively, and the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana–Champaign, USA, in 2015. He has worked as a Research Staff Member with the IBM T. J. Watson Research Center, from 2015 to 2019. He is currently an Assistant Professor with Hanyang University, South Korea. His main research interest includes efficient implementation of deep learning algorithms. He has received several research awards such as the DAC 2018 Best Paper Award and has actively contributed to the academic activities, such as the Technical Program Committee of DATE 2018–2020 (Co-Chair) and DAC 2018–2020, and Technical Committee (DiSPS) in IEEE Signal Processing Society.

YONG HO SONG (Member, IEEE) received the B.S. and M.S. degrees in computer engineering from Seoul National University, Seoul, South Korea, in 1989 and 1991, respectively, and the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, CA, USA, in 2002.

He is currently a Professor with the Department of Electronic Engineering, Hanyang University, Seoul, and the Senior Vice President of Samsung Electronics Company Ltd. His current research interests include system architecture and software systems of mobile embedded systems that further include SoC, NoC, multimedia on multicore parallel architecture, and NAND flash-based storage systems.

Dr. Song has served as a Program Committee Member of several prestigious conferences, including IEEE International Parallel and Distributed Processing Symposium, IEEE International Conference on Parallel and Distributed Systems, and IEEE International Conference on Computing, Communication, and Networks.