IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received November 7, 2021, accepted November 19, 2021, date of publication November 23, 2021,
date of current version December 3, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3130278

Buffer Management With Append-Only Data Isolation for Improving SSD Performance

JOONYONG JEONG[1], GYEONGYONG LEE[1], JUNGKEOL LEE[1], JUNGWOOK CHOI[1], (Member, IEEE), AND YONG HO SONG[1,2]

[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 research was supported by the MOTIE (Ministry of Trade, Industry & Energy (10080613) and KSRC (Korea Semiconductor Research Consortium) support program for the development of the future semiconductor device.

ABSTRACT The number of applications that can access a storage device simultaneously has increased as a result of the increase in storage capacity and the emergence of hyperscale environments. In multi-application environments, the request for append-only data to storage from applications such as log-structured merge-tree-based key-value (LSMKV) stores can negatively affect the storage-internal buffer hit ratio of other applications. This is because frequently re-accessed data can be evicted from the buffer via the intensive requests of append-only data that are rarely re-accessed. This degradation in the buffer hit ratio increases the storage access latency of applications. Herein, we propose a buffer management method to increase the buffer hit ratio of non-append-only data (or applications) in multi-application environments. The proposed method (1) defines large-sequential writes (that are not overwritten) and all reads on them as append-only input/output (I/O), (2) detects I/O, matching the access pattern of append-only data of LSMKVs, (3) allocates the append-only read/write requests into separate small buffer spaces, and (4) evicts the append-only data from the buffer when free buffer space is required. Because the proposed method stores append-only data of an LSMKV in buffer spaces with a limited size, it can increase the buffer hit ratio of applications that frequently re-access its data. Experimental results show that the proposed method can increase the buffer hit ratio of hot-data-intensive applications and the total buffer hit ratio by 31.72% and 20.06%, on average, respectively, in comparison to the existing buffer management techniques.

INDEX TERMS Buffer management, log-structured merge tree, multi-application, NAND flash storage.

I. INTRODUCTION

With the emergence of hyperscale environments such as Internet of Things (IoT) and cloud computing, the number of users that can access a storage device has increased [1]–[3]. Furthermore, as the processing performance of computing devices and the capacity of data storage devices have also increased, the number of applications that can be simultaneously operated by a user continues to grow. As a result, multiple applications can access one storage device simultaneously [1].

In these multi-application environments, the storage access pattern varies with the application; the application accesses the same data repeatedly or writes data that are rarely re-accessed to the storage. Data that are frequently accessed or can be potentially accessed in the near future are referred to as hot data while data displaying the opposite tendency are known as cold data. Heavy storage input/output (I/O) for cold data can increase the storage access latency of concurrently operating hot-data-intensive applications. This is because intensive I/O for data with low re-access frequency can evict hot data from the storage-internal buffer.

The I/O of cold data can frequently lead to hot data eviction in systems that use a limited data buffer space, such as solid-state drives (SSDs), which contain embedded RAM buffers. Because the buffer evicts data according to the page replacement policy in the event of inadequate free space, a small data buffer requires more frequent buffer evictions in comparison to a larger buffer. Thus, to load intensively requested cold data, a small buffer more frequently evicts buffer entries in comparison to a larger buffer. Consequently, hot data can be evicted from the buffer.

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

Log-structured merge-tree-based key-value (LSMKV) stores [4]–[7], that widely used NoSQL applications, are a type of cold-data-intensive applications, which request storage I/O for append-only cold data. LSMKVs store key-value pairs in storage devices in an immutable format. These key-value pair files (KV files) are managed in an append-only manner, wherein new KV files are added while the existing ones remain immutable. When KV files are written and merged, it leads to intensive read/write of append-only data [8], [9]. Because LSMKV systems use append-only management method for KV files, append-only data are requested to storage devices via various-sized read requests for scattered logical address space or large-sequential write requests which are not updated. Therefore, LSMKV applications have a low buffer hit ratio. Moreover, it can reduce the buffer hit ratio of other hot-data-intensive applications if their hot data are evicted due to intensive append-only data I/O.

To increase the buffer hit ratio, many existing buffer management methods predict the re-accessibility of the data of the buffer and preserve the hot data with high re-access probability in the buffer [10]–[17]. Data that are frequently or recently hit on the buffer [10], [12]–[17] or data requested in small sizes [11] are selected as hot data.

However, in the existing hot/cold data separation-based buffer management methods, hot data are evicted from the buffer when the I/O of append-only data from LSMKVs is intensively requested by large-sequential write and various-sized read operations. Conventionally, when the I/O of append-only data from LSMKVs is intensively requested for storage, hot data are evicted from the buffer before being re-accessed; otherwise, several instances of append-only cold data that are requested in small-random reads can be misinterpreted as hot data. This is because access-frequency/recency-based methods do not know the temperature of newly requested data and size-based methods interpret small-random read operations for append-only data as hot requests.

Herein, a buffer space separation and management method is proposed for a multi-application environment, including LSMKVs, to alleviate the degradation of the storage-internal-buffer hit ratio of hot-data-intensive applications resulting from the append-only data I/O of LSMKVs. The proposed method reduces the amount of hot data evicted by the append-only data I/O of the LSMKV application by (1) identifying I/O requests that match the I/O of append-only data from LSMKVs and (2) separating the buffer area where append-only and other data are stored.

The proposed method identifies the append-only data I/O of LSMKVs without additional information from the host; thus, hot data eviction due to LSMKV I/O can be reduced even if hot-data-intensive applications and their data access patterns are different. Our method compares the access pattern of incoming I/O and append-only data I/O of the LSMKV to detect whether the incoming data I/O is requested for append-only data. The append-only data of the LSMKV are written by large-sequential write requests that are not updated; these data are read in various sizes and patterns. Large-sequential writes can be identified by monitoring the request size of the incoming I/O command. However, not all large-sequential writes are append-only data, and read requests for append-only data cannot be distinguished by the request size. Additionally, when the data requested by large-sequential writes are updated, the data of the corresponding logical address should not be classified as append-only data.

Tracing the access history of a logical address is a method used to determine whether the data of the requested logical address are append-only. Overwrite does not occur in the logical address where append-only data are stored because update requests at the host side are executed by overwriting data in a storage device. To record the access history of logical addresses, the proposed method uses an additional mapping table, a large write & append-only data check table (LAT), which maps the logical address, request size, and overwrite check bit. Because the LAT is stored in the buffer, a large LAT can reduce the buffer hit ratio and available space in the buffer. The proposed LAT decreases the size of the address information stored in the LAT by mapping addresses to coarse-grained granularity.

To limit the amount of non-append-only data evicted from the buffer due to append-only data I/O, the proposed method loads append-only and non-append-only data into separate buffer spaces. The proposed buffer is divided into three regions, namely a buffer space for append-only write data (BAW), buffer space for append-only read data (BAR), and the remaining area for data excluding append-only data (small or re-accessed data buffer space, SORB). Because the append-only data of the LSMKV are not updated after being written into the storage, the probability of the LSMKV being hit in the buffer is lower than that of other applications. Therefore, while a limited size is allocated to BAW and BAR, almost the entire buffer size can be allocated to SORB.

In addition, a buffer eviction algorithm is presented in this study to select the area of the buffer entry for eviction from the proposed buffer areas in the event of insufficient free space in the buffer. In the proposed method, the append-only data are allocated only to BAW or BAR. Because BAW and BAR have a limited maximum size, eviction must be performed even if there is free space in the entire buffer if either one reaches the maximum size. In addition, because append-only data are less likely to be re-accessed, BAW and BAR should be prioritized over SORB when there is no free space in the entire buffer.

The remainder of this paper is organized as follows. Section 2 presents the background information regarding the proposed method. In Section 3, previously proposed buffer management schemes (page replacement algorithms) are discussed. Section 4 describes the proposed method and section 5 presents the experimental setup and results. Finally, the conclusions are presented in Section 6.

II. BACKGROUND INFORMATION

A. LOG-STRUCTURED MERGE-TREE-BASED KEY-VALUE STORES

LSMKVs are a traditional NoSQL database [4]–[7], [18]. LSMKVs store KV pair data in an append-only manner. In an LSMKV, all key-value writes are buffered in the memory (specifically, the main memory of the host side), and the data in memory are flushed into the storage via large-sequential writes [19].

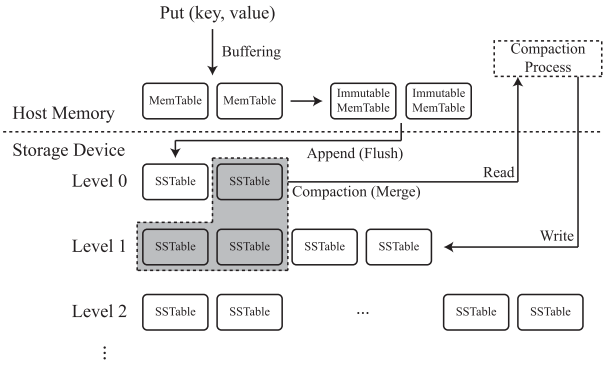


FIGURE 1. KV pair write sequence of LSMKV store.

In an LSMKV, the inserted KV data are stored in the form of an immutable file containing multiple KV pairs. These immutable KV files stored in the storage are not updated; the incoming KV pairs are appended into the storage as a new KV file. In several LSMKVs, an immutable KV file stored in storage is referred to as an SSTable.

SSTables are managed in a hierarchical structure; the more recently entered SSTables are stored at higher levels in the hierarchy. Each layer has a limited size and when the size of one layer exceeds the limit, an interlayer data merge operation (called compaction) is performed. Compaction merges the SSTables of the layer that exceed the limit with those of the lower layer and moves them to the lower layer. At the end of the compaction process, the input SSTables are deleted and the compacted output SSTables are written into the storage. Fig. 1 demonstrates the write process of the KV pair of an LSMKV. When the lookup of the value of a given key is requested, the LSMKV searches for the given key by identifying the key range of the SSTable, starting from the highest layer [20], [21]. Fig. 2 illustrates the cascading read requests induced by such lookup request in the LSMKV.

Figure 3 depicts the requested storage I/O size of RocksDB [5], a widely used LSMKV. Table 1 summarizes the workload used in the experiment for Fig. 3. As shown in Fig. 3, the storage access pattern of the LSMKV is primarily large-sequential write and large-sequential or various-sized read, induced by the write and read of the SSTable, respectively. Large-sequential reads and writes frequently occur in the reads of SSTables to be compacted and the writes of SSTables resulting from compaction, respectively. Furthermore, large-sequential and small-random reads occur frequently during SSTable traversal for a given key search.

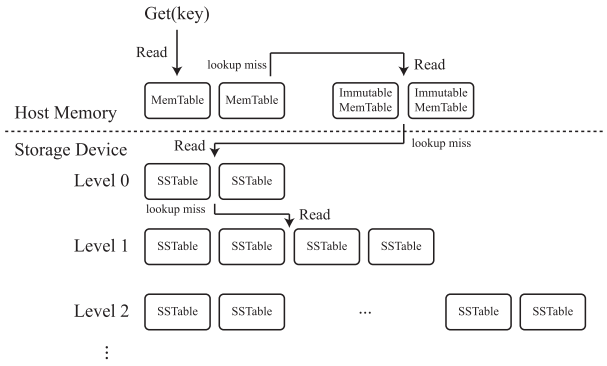


FIGURE 2. Cascading read during the lookup sequence of LSMKV store.

Because the stored SSTables are input via the append-only method, the possibility of re-access is less, thus leading to a considerably less buffer hit ratio for the LSMKV.

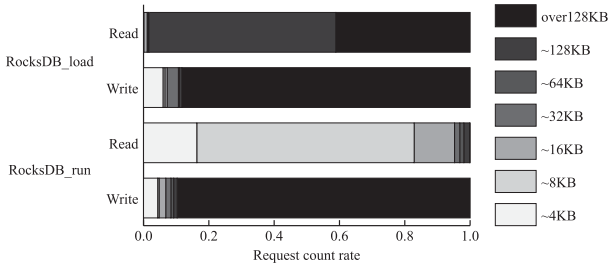


FIGURE 3. Request size distribution of RocksDB.

B. SOLID STATE DRIVE

SSDs with built-in RAM are widely used as secondary storage devices [11]. A modern SSD comprises a host interface, DRAM, embedded processor, NAND flash controller, and NAND flash chips [22]–[24]. The read/write unit of the NAND flash is a flash page. However, modern SSDs can access multiple NAND flash chips in parallel by configuring NAND chips to be multi-channel and multi-way [25], [26]. Fig. 4 presents an SSD with a two-channel and two-way configuration. Because each channel of an SSD is configured independently, flash chips at different channels can be accessed in parallel. In contrast, NAND flash chips, configured with different ways within a channel, transfer data through the same channel. Therefore, different ways of SSD are accessed in parallel by pipelining the data transfer of the channel and write operation of the NAND cell.

A NAND flash device must erase data before writing the data as required by the recording method of the medium. The erase operation deletes all pages of the victim NAND flash block to switch the NAND block to a writable state. Because the erase unit of the NAND flash is larger than the write unit, a flash block can contain both invalid and valid pages. The valid pages of a victim NAND flash block must be preserved; therefore, valid pages are copied to a recordable block when a victim NAND flash block is erased.

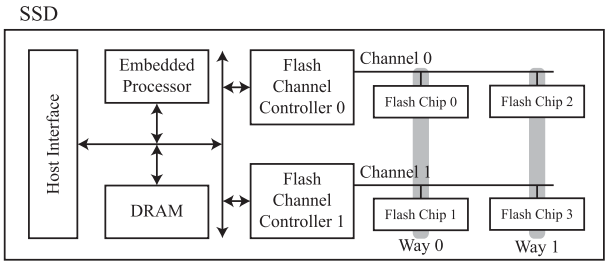


FIGURE 4. Overall structure of SSD with a two-channel two-way configuration.

Previously, data deletion from the host was not requested as a command to the storage device. However, if the deleted data remain valid in the storage device, it may lead to frequent garbage collection in SSDs. Therefore, some modern SSDs support the trim function [27]–[31]. Trim is a command that passes the logical address, which is no longer used by the host, to the storage and invalidates the logical address area in the storage. We assumed that the SSD in this study supports the trim function.

The RAM inside the SSD is used as a data buffer and stores information such as the metadata of the flash translation layer to manage NAND flash storage [24]. If the data of the requested I/O command exist in the data buffer, the storage returns the requested data by accessing only the DRAM (not NAND flash). Because the access latency of NAND flash chips is slower than that of DRAM, buffer hits can improve the response time of I/O commands for storage.

III. RELATED WORK

A. BUFFER MANAGEMENT SCHEME FOR SOLID STATE DRIVE

In systems that use SSDs as storage devices, the storage access latency can be decreased by reducing the number of NAND flash accesses required to process data I/O commands. The number of accesses to the NAND flash can be reduced through several buffer management methods for NAND flash storage. These methods (1) increase the hit ratio of the RAM buffer [10]–[17], (2) reduce the number of dirty data write operations from the buffer to the NAND [12]–[17], [32], [33], and (3) reduce the number of valid page migrations performed in the NAND block erase operation [32], [34]–[36]. The I/O and eviction granularity of traditional buffer management for SSDs are performed in units of a page or block, which are the same as those of NAND flash.

Page-based techniques are more suitable than block-based techniques for increasing the buffer hit ratio or reducing the number of dirty data writes from the buffer to the NAND flash. This is because, in hot/cold prediction or clean/dirty separation of buffer entries, the accuracy of prediction/separation increases as the buffer management granularity decreases [37]. Thus, for a larger buffer management granularity, the hot/cold or clean/dirty data are more likely to be mixed within one buffer entry.

Various existing buffer management methods have been proposed based on least recently used (LRU) algorithm, which is a traditional page replacement method. The LRU evicts the longest inaccessible data to increase the hit ratio based on temporal locality. Chang proposed a two-level LRU [10] based on the concept of hot-cold separation to separately manage frequently accessed logical address areas. The two-level LRU inserts the incoming entries into the candidate LRU. A re-accessed entry in the candidate LRU is promoted to the hot LRU list. If there is no free area in the hot LRU list, the LRU end entry of the hot LRU is demoted to candidate LRU. Thus, the two-level LRU can reduce the number of hot data events owing to the input of cold data.

The virtual-block-based buffer management scheme (VBBMS) [11] is a buffer management technology that uses a size-based hot-cold separation concept. VBBMS divides the buffer into two regions, namely, the sequential request service region (SRSR) and random request service region (RRSR), wherein large-sequential requests and small -random requests are assigned, respectively. Re-accessed buffer entries in SRSR are migrated to RRSR. In addition, the VBBMS introduces a virtual block as a new buffer management granularity. The virtual block consists of multiple pages thus it is larger than one page and smaller than one block. SRSR and RRSR can have different virtual block sizes.

Clean-first LRU (CFLRU) [33] manages the buffer as an LRU list consisting of clean-first and working regions. The clean-first region indicates the window size w entries from the LRU end. The primary idea of CFLRU is that when buffer eviction is required, clean pages are preferentially selected as victim entries in the clean-first region. A clean page of the buffer is not written to the NAND flash when it is evicted from the buffer. Therefore, by evicting clean pages before dirty pages, the number of NAND flash accesses required for buffer eviction operations can be reduced.

Cui *et al.* proposed a probabilistic triplicate LRU (PTLRU) to alleviate hot clean page eviction and cold dirty page retention in CFLRU [15]. PTLRU manages the buffer with three LRU lists, that is, cold clean LRU list (LC), cold dirty LRU list (LD), and mixed LRU list (LH), the size of which is not fixed. In PTLRU, if a reference to the data requested to the buffer is hit in the buffer, the referenced entry is moved to the MRU position in the LH list. If the I/O request is missed in the buffer, the PTLRU loads data into the buffer space; if free space exists in the buffer, the data are inserted into the LC list. However, if there is no free space in the buffer, PTLRU preferentially evicts the LRU end entry of the LC list. If an entry in the LC list (cold clean page) does not exist in the buffer, the LD list entry (cold dirty page) is evicted based on the predefined probability.

However, existing buffer management methods are not suitable for preventing hot data eviction from the buffer when multiple applications, including the LSMKV, access the storage simultaneously. First, access frequency-based hot/cold separation techniques are not suitable for reducing hot data eviction resulting from the I/O of the LSMKV. In situations

where append-only data I/O is requested intensively, hot data are evicted before being re-accessed from the buffer (Fig. 5). For example, in a two-level LRU, the entries of candidate LRU that have not yet been re-accessed may be evicted because of the intensive I/O of cold data, even though the entries are highly likely to be re-accessed. Second, clean-first-based buffer management methods can effectively reduce the number of flash write operations generated by dirty data eviction. Nevertheless, because many existing clean-first-based methods do not include hot/cold separation considering the storage access patterns of append-only data I/O, the proposed clean-first methods have similar limitations similar to those of existing hot/cold separation methods. Finally, the size-based hot/cold separation technique encounters difficulties in mitigating hot data eviction when small reads for append-only data are intensively requested for storage (Fig. 6). For example, in a multi-application environment with the LSMKV, the VBBMS may allocate many cold small-random reads to the RRSR (region where hot data are stored) and allocate several hot and cold large-sequential I/Os to the same buffer space, particularly SRSR.

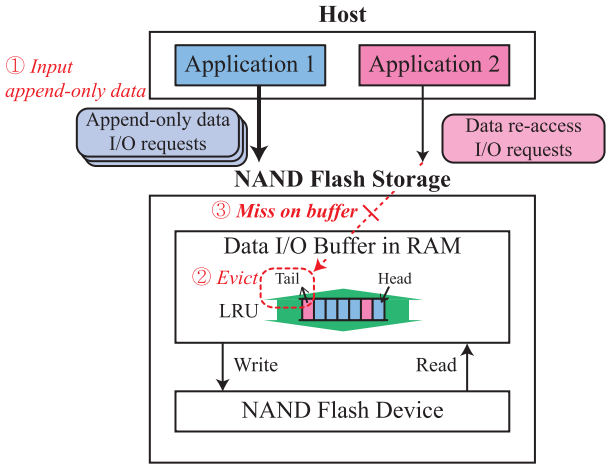


FIGURE 5. Storage access performance degradation scenario of the access-frequency-based buffer management method when multiple applications access one NAND flash storage.

IV. MAIN IDEA

In this study, we propose a buffer management method to alleviate the eviction of non-append-only data caused by the I/O of append-only data from the LSMKV application. Because the append-only data of the LSMKV are not updated, append-only data write requests are not hit on the buffer. Read requests for append-only data are also rarely hit on the buffer because many reads are requested as large-sequential patterns for consecutive logical addresses or small-random patterns for a scattered address space. When data with a low hit probability on the buffer are requested intensively, data with a high probability of being re-accessed in the buffer may be evicted from the buffer. If the evicted data are re-accessed, the buffer hit ratio of the application requesting

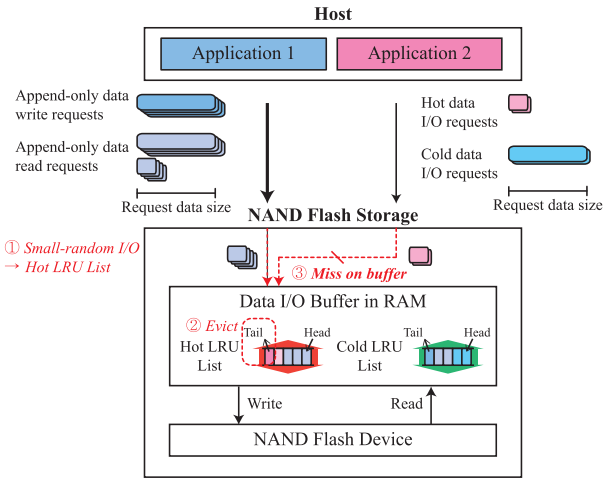
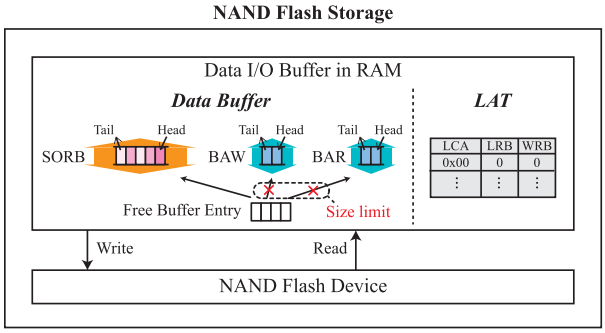


FIGURE 6. Storage access performance degradation scenario of the size-based buffer management method when multiple applications access one NAND flash storage.

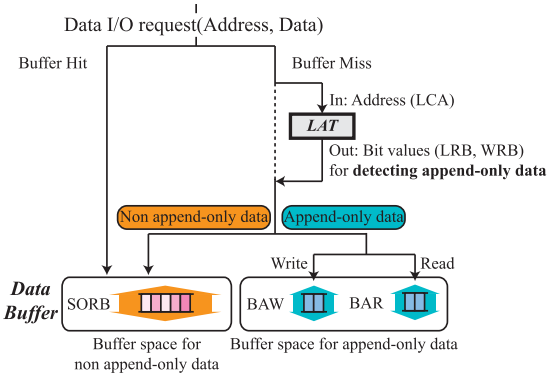
re-access can decrease, and the storage access latency of the application may deteriorate. Therefore, the proposed method allocates I/O requests that match the storage access pattern of the append-only data I/O of the LSMKV to a separate buffer space. To separate the buffer space for append-only data from non-append-only data, the proposed method (1) identifies the I/O matching the append-only data I/O pattern of the LSMKV from the incoming I/O requests, (2) divides the buffer space into three spaces (SORB, BAW, and BAR), and (3) allocates append-only writes and reads to BAW and BAR, respectively. Moreover, an LAT is proposed to select the I/O for append-only data from incoming I/O requests. The I/O of append-only data for the LSMKV is requested for storage through large-sequential writes (without re-access for stored data) and large-sequential or small-random reads. To verify whether the incoming write is a request for re-access to the stored data, the LAT maps the logical address, requests size, and writes the re-access history. In addition, because the proposed method allocates I/O requests for append-only data to BAW and BAR, if free space is insufficient in BAW and BAR, the buffer entry must be evicted even if there is free space in the entire buffer. Section 4.A describes the structure of the proposed buffer and LAT; Section 4.B describes the read/write/eviction method of the proposed buffer. Fig. 7 illustrates the overall structure of the proposed buffer management method and a brief buffer access sequence.

A. BUFFER SPACE SEPARATION TECHNIQUE

Based on the proposed method, a buffer is classified into a data buffer space and LAT storage space. The data buffer space is further divided into SORB, BAW, and BAR. BAR and BAW represent the read and write buffer for append-only data, respectively. To utilize the internal parallelism of NAND flash storage, the maximum size of BAR and BAW is the smallest I/O size that can achieve the maximum internal parallelism in NAND flash storage. For example, for the SSD



(a) Brief architecture of proposed buffer



(b) Buffer access sequence of proposed buffer management technique

Abbreviations
- LAT: Large write & append-only data check table
- LCA: Logical chunk address
- LRB: Large-size requested bit
- WRB: Write re-access bit
- SORB: Small or re-accessed data buffer space
- BAR: Buffer space for append-only read data
- BAW: Buffer space for append-only write data

FIGURE 7. Brief architecture and buffer access sequence of the proposed buffer management method. The I/O buffer in the RAM of the proposed method stores data and LAT.

depicted in Fig. 4, the maximum size of each of BAR and BAW is $Size_{page} \times Count_{channel} \times Count_{way} = 4pages$. SORB stores all data except append-only data, i.e., data requested by a small write or update request (re-write for the logical address allocated to valid data). The size of the SORB is $Size_{TotalBuffer} - Size_{BAW} - Size_{BAR}$; the entire buffer space of the buffer can be used as SORB when I/O for append-only data is rarely requested. The LRU constitutes the page replacement policy of SORB, BAW, and BAR, but other existing page replacement policies can be applied. Therefore, the proposed method can be considered as an extension of the existing buffer management method.

In this study, write re-access (update) does not occur in append-only data after the data are written to the buffer or storage. Because the append-only data of the LSMKV are written by large-sequential requests, the proposed method interprets large-sequential write data that have not been updated as append-only data.

To distinguish between I/O commands for append-only data from the I/O data requests to the storage, we propose an LAT that records the requested I/O size of logical addresses and update history. If the size of a write request is larger than the predefined threshold, the proposed method considers the requested I/O as a large-sequential write. To indicate that the write command requested in the address space is a large-sequential write, the LAT uses a large requested bit (LRB). If the requested write is a large-sequential write, the LRB of the LAT for the requested logical address is set at 1. Furthermore, we can determine whether the logical address has been updated by recording the write re-access to the requested logical address. The proposed method uses a write re-accessed bit (WRB) to determine whether the logical address of the requested I/O command has been updated. If the LRB mapped to the requested logical address is 1, then a large-sequential write has been performed on the logical address. Therefore, a write request to a logical address with an LRB of 1 represents a write re-access request. The proposed method sets the WRB of the write- re-accessed logical address to 1. If the WRB mapped to the logical address of the requested read command is 1, the requested command is a read request for the updated data, i.e., non-append-only data; otherwise, it is a read request for append-only data. The initial value of the proposed LRB and WRB is 0. The larger the size of the LAT, the smaller the buffer space used for data buffer. To secure a larger data buffer space while using LAT, the proposed method reduces the size of LAT through chunk unit addressing. The chunk size of the proposed method is 1 MB by default, which can be adjusted. The size of the LAT was calculated as (1). Each table entry is accessed by logical chunk address (LCA).

$$Size_{LAT} = \left(\frac{Size_{SSD}}{Size_{chunk}}\right) \times (WRB + LRB) \quad (1)$$

B. BUFFER SPACE MANAGEMENT TECHNIQUE

The buffer space to record data is selected from SORB, BAR, and BAW by checking the logical address of the incoming data I/O request and the LAT item corresponding to the address. A detailed description of buffer read, write, and eviction operations is given below.

For a requested write command, the proposed method monitors the data request size and logical address. The logical address of the requested data is converted into a LCA (line 1 of Algorithm 1). When the requested write is hit in the buffer, WRB is set at 1, mapped to the requested LCA, and data are loaded into the SORB area. If a buffer hit does not occur, the buffer checks the LAT items mapped to the corresponding LCA; LRB, and WRB (line 6 of Algorithm 1).

If the data size of the write request is less than the predefined threshold and LRB is 0, the proposed method writes data into the SORB area (*(1)* depicted in Fig. 8, lines 12–15 of Algorithm 1). Unlike the BAW and BAR areas, the SORB area can be allocated up to the entire buffer size. When data I/O to the SORB area is intensively requested, the proposed method increases the size of the SORB area by evicting the data from the BAW and BAR areas (lines 4–14

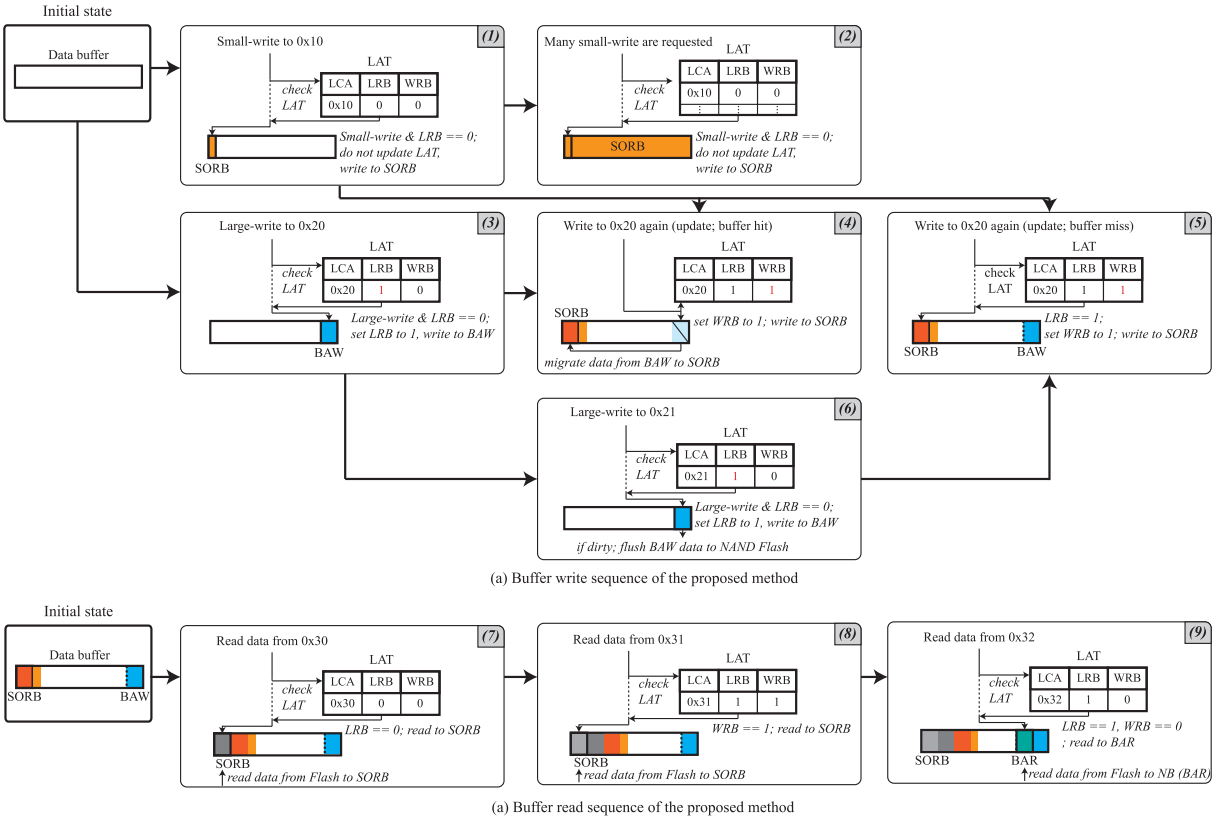


FIGURE 8. **(a) Buffer write sequence and (b) buffer read sequence of the proposed method.**

of Algorithm 5). The proposed buffer eviction sequence is presented in Algorithms 3, 4, and 5, and Fig. 9, 10, 11.

If the data size of the write request exceeds the predefined threshold and LRB and WRB are both 0 (lines 7 and 8 of Algorithm 1), then the LRB is updated to 1 (line 9 of Algorithm 1) and the data are written to the BAW area (*(3)* and *(6)* in Fig. 8, line 11 of Algorithm 1). In the proposed method, because all small writes are recorded in the SORB area, the buffer need not refer to the LAT for selecting a space to write data. However, to check whether the requested small write is an update command for append-only data, the LAT check process is required (line 6 of Algorithm 1).

If a buffer hit occurs or the LRB mapped to the address of the requested write is 1, the requested write is a re-access to the data written into the storage(*(4)*, *(5)* in Fig. 8, and lines 2–4, 12–15, and 21–24 of Algorithm 1). If the LRB corresponding to the requested logical address is 1, WRB is updated to 1 and the data are written into the SORB area (*(5)* of Fig. 8, lines 12–15 and 21–24 of Algorithm 1). If the requested I/O is hit in the buffer, the data are migrated to the SORB area (*(4)* in Fig. 8, and line 2–4 of Algorithm 1).

For a requested read command, the proposed method monitors the logical address. If the LRB corresponding to the requested logical address is 1 and WRB is 0 (line 6 of Algorithm 2), the data are loaded into the BAR area (*(9)* in Fig. 8, lines 7–9 of Algorithm 2). If LRB is 0 or the WRB area is 1, the data are loaded into the SORB area (*(7)*, *(8)* of Fig. 8, and lines 10–13 of Algorithm 2).

Before loading the data requested by a read or write command into the memory, the buffer checks its free space (lines 10, 14, 19, and 23 of Algorithm 1; lines 7 and 11 of Algorithm 2; and Algorithm 3, 4, and 5). When there is insufficient space in the buffer to load the requested I/O, the developed buffer selects an area to evict data from the three buffer spaces, i.e., SORB, BAR, and BAW. Fig. 9, 10, 11 and Algorithm 3, 4, 5 show the buffer eviction method of the proposed method.

Even if there is free space in the entire buffer, if read or write of append-only data is requested when there is no free space in the BAR or BAW area, the proposed method evicts the data of BAR or BAW without loading the data into the available space of the buffer (lines 13–16 of Algorithm 3 and lines 19–21 of Algorithm 4). This is because the goal of the proposed method is to increase the buffer hit ratio for non-append-only data by reducing the amount of non-append-only data evicted from the buffer due to I/O of append-only data. Therefore, the append-only data are allocated only to the BAR and BAW areas with a limited maximum size, unlike non-append-only data.

In the proposed method, a large write request, which is not an update for the stored data, is allocated to BAW (lines 7–11 of Algorithm 1). However, if there is inadequate free space

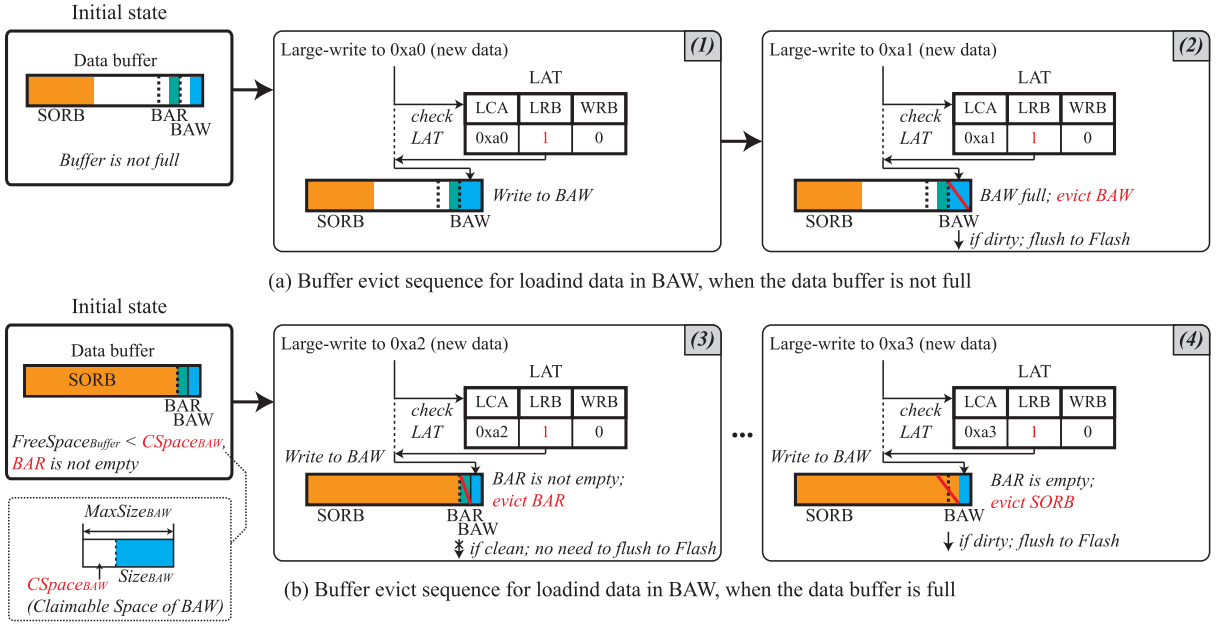


FIGURE 9. Buffer eviction sequence of the proposed method for loading data in BAW. (a) Buffer eviction sequence when the data buffer is not full. (b) Buffer eviction sequence when the data buffer is full.

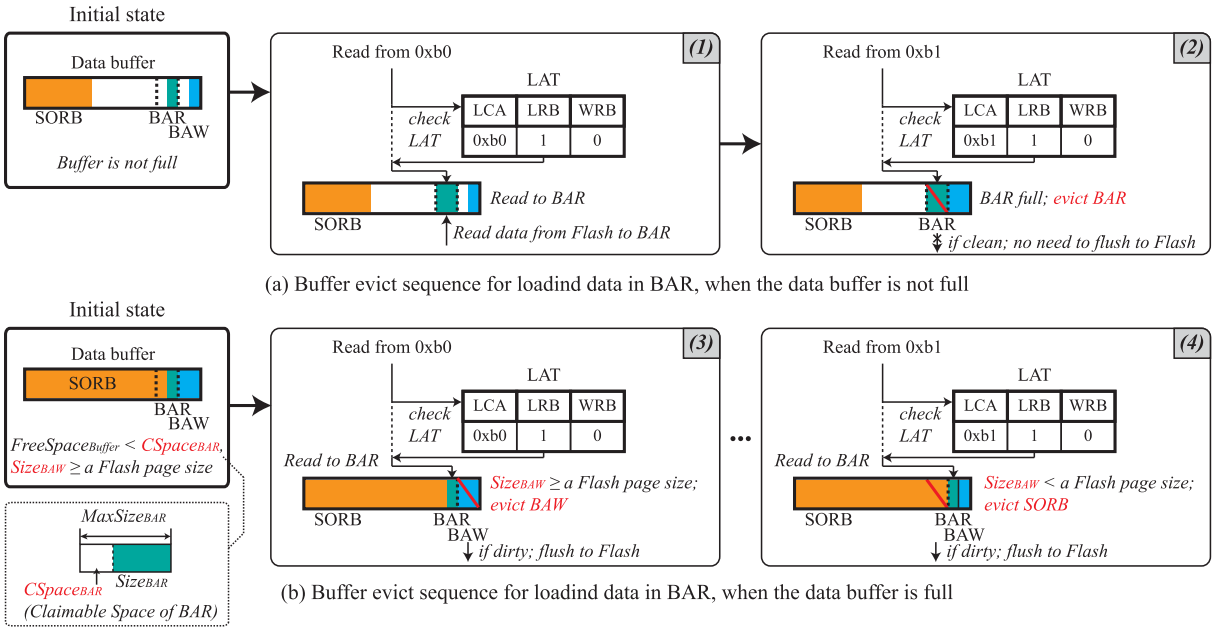


FIGURE 10. Buffer eviction sequence of the proposed method for loading data in BAR. (a) Buffer eviction sequence when the data buffer is not full. (b) Buffer eviction sequence when the data buffer is full.

in the BAW area, the proposed method evicts the data in this area even if there is free space in the entire buffer, and writes the incoming data to the BAW area (*(1)* and *(2)* shown in Fig. 9, and lines 13–16 of Algorithm 3). The maximum size of the BAW and BAR areas is limited to the maximum internal parallelism I/O size of SSD. In contrast, the maximum size of the SORB area corresponds to the total data buffer size. Therefore, when a large amount of data are loaded into the SORB area, the free space of the buffer may be insufficient; accordingly, the BAW area may not be extended maximally. In this case, when a large-sequential write is requested from the BAW, the flash write required in the data eviction process of the BAW area may not be performed by utilizing the internal parallelism of the SSD. If data write to BAW is requested when it cannot be expanded to its maximum size because of insufficient free space in the buffer, the buffer of

Algorithm 1 Buffer Write Sequence of the Proposed Method

Input: logical block address *LBA*,
total data size of host request *reqSize*,
memory-IO data *data*,
memory-IO size $Size_{data}$
Output: none

```
1:  LCA = (LBA/ChunkSize);
2:  if BufferHit then
3:      set WRB to 1 for given LCA to LAT;
4:      write data to SORB;
5:  else
6:      get WRB and LRB for given LCA from LAT;
7:      if reqSize > threshold then
8:          if (WRB == 0)&(LRB == 0) then
9:              set LRB to 1 for given LCA to LAT;
10:             BufferEvict(Size_data, BAW);
11:             write data to BAW;
12:         else
13:             set WRB and LRB to 1 for given LCA to LAT;
14:             BufferEvict(Size_data, SORB);
15:             write data to SORB;
16:         end if
17:     else
18:         if LRB == 0 then
19:             BufferEvict(Size_data, SORB);
20:             write to SORB;
21:         else
22:             set WRB to 1 for given LCA to LAT
23:             BufferEvict(Size_data, SORB)
24:             write to SORB
25:         end if
26:     end if
27: end if
```

Algorithm 2 Buffer Read Sequence of Proposed Method

Input: logical block address *LBA*,
memory-IO size $Size_{data}$
Output: memory-IO data *data*

```
1:  LCA = (LBA/ChunkSize);
2:  if BufferHit then
3:      return data;
4:  else
5:      get WRB and LRB for given LCA from LAT;
6:      if (LRB == 1)&(WRB == 0) then
7:          BufferEvict(Size_data, BAR);
8:          read data from NAND to BAR;
9:          return data;
10:     else
11:         BufferEvict(Size_data, SORB);
12:         read data from NAND to SORB;
13:         return data;
14:     end if
15: end if
```

Algorithm 3 Buffer Eviction Sequence of the Proposed Method for Loading Data in BAW

Input: memory-IO data size $Size_{data}$,
destination buffer space of I/O request *Dest*
Output: none

```
1:  //CSpace_BAW == Claimable buffer space of BAW
2:  //CSpace_BAW = MaxSize_BAW − Size_BAW
3:  if Dest == BAW then
4:      if FreeSpace_Buffer < CSpace_BAW then
5:          while FreeSpace_Buffer < Size_data do
6:              if Size_BAR > 0 then
7:                  flush BAR;
8:              else
9:                  evict victim from SORB;
10:             end if
11:         end while
12:     else
13:         while FreeSpace_BAW < Size_data do
14:             evict victim from BAW;
15:         end while
16:         Deadline_BAW = 0;
17:     end if
18: end if
```

the proposed method first flushes the BAR area, and evicts the data of SORB area *((3)* and *(4)* in Fig. 9 and lines 3–11 of Algorithm 3). In Fig. 9, 10 and Algorithms 3 and 4, $CSpace_x$ represents the amount of buffer space that buffer x can claim currently. The size of the claimable buffer space of x is derived from the difference between the maximum size of the buffer space x and the size of the buffer space currently allocated to x. $Deadline_{BAW}$ in line 16 of Algorithm 3 is used in the eviction of the SORB area.

If the LRB for the requested read is 1 and WRB is 0, the proposed method determines that the requested read is for append-only data and allocates incoming read to the BAR area (lines 6–9 of Algorithm 2). If the available space in the BAR area is insufficient, the proposed method evicts the data in the BAR area, reads the data from the flash, and loads it in the BAR area *((2)* in Fig. 10, lines 7–9 of Algorithm 2, and lines 19–21 of Algorithm 4). As mentioned in the eviction process of the BAW area, the buffer space of the SORB area can be equal to the entire buffer size. Therefore, when a large amount of data are loaded into the SORB area, the BAR area may not be extended maximally. In this case, if a large-sequential read is requested from BAR, the internal parallelism of the SSD may not be highly utilized when NAND flash read due to buffer miss is processed. When BAR cannot be expanded up to its maximum size owing to insufficient free space in the buffer, the proposed method first evicts BAW and then evicts SORB *((3)*, *(4)* in Fig. 10, and lines 3–17 of Algorithm 4).

It is assumed that the update frequency of the data stored in the SORB area and the append-only data stored in the BAW

area differ from each other; data stored in the BAW area are less frequently updated than those in the SORB area. If data with different update frequencies are stored in the same flash page, (1) valid page copies may be frequently occurred during the garbage collection process of NAND flash, which may increase the write amplification factor and (2) the storage-internal-buffer may be polluted. To avoid writing append-only and non-append-only data to the same flash page, the BAW area is evicted in units of flash pages, which is the minimum I/O unit of NAND flash storage (line 6 of Algorithm 4 and line 6 of Algorithm 5). If the size of data in the BAW area is larger than that in a NAND flash page, the data in the BAW area are evicted (*(3)* in Fig. 10 lines 6–8 of Algorithm 4).

If the size of data in the BAW area is less than that in a flash page, the data in the SORB area are evicted (*(4)* in Fig. 10, line 15-16 of Algorithm 4). When the size of data in the BAW area is less than the size of a flash page, the data in the BAW area cannot be evicted and may reside in the buffer (*(4)* in Fig. 10). Append-only data residing in the buffer are less likely to be hit, which reduces the size of the available buffer space, thereby reducing the buffer hit rate. If the data in the BAW area cannot be evicted from the buffer because its size is less than a flash page size, the size of data input to the buffer is monitored to determine when BAW data will be evicted (line 10 of Algorithm 4). If BAW data are not flushed until data are input for the entire data buffer size, the data in the BAW area are evicted (lines 11–13 of Algorithm 4).

If I/O is requested to the SORB area when there is no free space in the buffer, the buffer area from SORB, BAR, or BAW is first selected for eviction (Algorithm 5). If data exist in the BAR area, the buffer evicts them with the highest priority (*(1)* in Fig. 11, lines 3-5 of Algorithm 5). This is because evicting the data in the BAR area will not likely increase the latency of I/O requests waiting to be processed. Because the BAR area is used as a read buffer, the BAR data are clean, except when a read hit for dirty data occurs. Therefore, data write to flash is not required during eviction. When there are no data in the BAR area, the eviction priority between the BAW and SORB areas is determined in a similar manner as the eviction sequence for loading data in BAR (*(3)*, *(4)* of Fig 10 and *(2)*, *(3)* of Fig. 11; lines 4–17 of Algorithm 4 and lines 6–17 of Algorithm 5).

V. EXPERIMENTS

To evaluate the proposed method, the target application-specific buffer hit ratio ($HitRatio_{TA}$), total buffer hit ratio ($HitRatio_{Total}$), target application-specific storage access latency ($latency_{TA}$) and total storage access latency ($latency_{Total}$) were measured. $HitRatio_{TA}$ and $latency_{TA}$ represent the buffer hit ratio and storage access latency of the target application in an environment where multiple applications access the buffer, respectively. The storage access workload other than the LSMKV is the target application in this study (Table 1). This is because the objective of this study is to alleviate the decrease in the buffer hit ratio of other applications

Algorithm 4 Buffer Eviction Sequence of the Proposed Method for Loading Data in BAR

Input: memory-IO data size $Size_{data}$, destination buffer space of I/O request *Dest*
Output: none

```
1:  //CSpace_BAR == Claimable buffer space of BAR
2:  //CSpace_BAR = MaxSize_BAR − Size_BAR
3:  if Dest == BAR then
4:      if FreeSpace_Buffer < CSpace_BAR then
5:          while FreeSpace_Buffer < Size_data do
6:              if Size_BAW ≥ Size_NANDPage then
7:                  Deadline_BAW = 0;
8:                  evict victim from BAW;
9:              else if Size_BAW > 0 then
10:                 Deadline_BAW += Size_data;
11:                 if Deadline_BAW ≥ Size_Buffer then
12:                     Deadline_BAW = 0;
13:                     flush BAW;
14:                 end if
15:             end if
16:             evict victim from SORB;
17:         end while
18:     else
19:         while FreeSpace_BAR < Size_data do
20:             evict victim from BAR;
21:         end while
22:     end if
23: end if
```

Algorithm 5 Buffer Eviction Sequence of the Proposed Method for Loading Data in SORB

Input: memory-IO data size $Size_{data}$, destination buffer space of I/O request *Dest*
Output: none

```
1:  //UsableSpace_x = MaxSize_x − Size_x
2:  if Dest == SORB then
3:      while FreeSpace_Buffer < Size_data do
4:          if Size_BAR > 0 then
5:              flush BAR;
6:          else if Size_BAW ≥ Size_NANDPage then
7:              Deadline_BAW = 0;
8:              evict victim from BAW;
9:          else
10:             if Size_BAW > 0 then
11:                 Deadline_BAW += Size_data;
12:                 if Deadline_BAW ≥ Size_Buffer then
13:                     Deadline_BAW = 0;
14:                     flush BAW;
15:                 end if
16:             end if
17:             evict victim from SORB;
18:         end if
19:     end while
20: end if
```

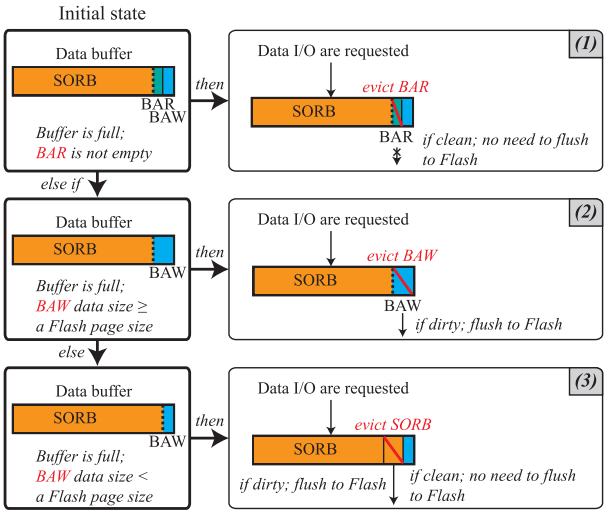


FIGURE 11. Buffer eviction sequence of the proposed method for loading data in SORB.

that simultaneously access the storage owing to the I/O of append-only data for the LSMKV application.

A. EXPERIMENTAL SETUP

1) WORKLOADS

An experiment was performed using an in-house trace-driven simulator, which is an extension of FlashSim [38], to (1) implement various buffer management algorithms, (2) use storage I/O access traces of various applications, and (3) measure the buffer hit ratio and storage access latency for each application. Table 1 lists the details of the workloads used in the experiment.

RocksDB [5], a widely used LSMKV, was selected as the LSMKV application. The storage access workload of RocksDB was recorded by running YCSB [39], which is a widely used benchmark tool for cloud systems and databases. The Zipfian distribution random order (zipf), which is often used for the performance evaluation of various KV stores, was selected as the input key pattern of YCSB [40]–[44]. ''Load'' in YCSB stores a predefined number of KV pairs in the database. In addition, ''Run'' reads half of the predefined number of KV pairs and updates the remaining. Note that updating a KV pair is different from updating the data stored in the storage. Updating a KV pair involves inserting a different value for the same key. Meanwhile, updating the data stored in the storage involves rewriting data to the same logical address. In Table 1, Rocks_load represents the storage access workload when 200,000,000 key-value pairs are newly loaded into RocksDB and Rocks_run represents the storage access trace of read and update for each 50,000,000 KV pairs. As mentioned in the performance benchmark of RocksDB [5], the load of RocksDB is approximately 10 times faster than read and update (overwrite); moreover, the storage access collection of Rocks_load is completed in less time than that of Rocks_run. In Rocks_load, intensive storage I/O (specifically, large-sequential writes) occurred in a shorter time in comparison to Rocks_run. In Rocks_run, more small-random or large-sequential reads were recorded than Rocks_load.

The storage access workloads of the target application were collected from the Microsoft Research (MSR) Cambridge workload [45]. The workloads (W1 to W14) used to evaluate the performance of the proposed method were constructed by synthesizing the storage access trace of the LSMKV application and that of the target application (Fig. 12). For Rocks_load, storage I/O trace collection was completed in a shorter time than Rocks_run. Therefore, for W1 to W7, only a fraction of the storage I/O of the target application was used for synthesis. In contrast, for W8 to W14, all storage accesses of the target application were used for synthesis.

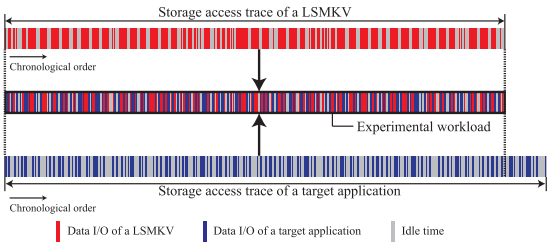


FIGURE 12. Conceptual diagram of the configuration of experimental workloads.

2) ALGORITHMS

The characteristics of the algorithms used in the experiment are as follows. The two-level LRU and PTLRU manage data with high-access and low-access frequencies as separate LRUs. While the two-level LRU manages the buffer with fixed-length LRU lists, the maximum length of LRU lists constituting the PTLRU is not determined. Therefore, the PTLRU may store more hot data than the two-level LRU. However, if the LRU list of the PTLRU that stores hot data becomes large, the maximum size of the list that stores input data reduces. Hence, the input data with the possibility of re-access may be evicted from the buffer, in comparison to the two-level LRU. VBBMS and the proposed method allocate the requested large- and small-sized data to different LRU lists. In comparison to VBBMS, the proposed method allocates 1) large-sequential writes without update or overwrite and 2) read requests for append-only data to small LRU lists.

3) PARAMETER SETTING

A buffer size of 8–128 MB was used in the experiment, corresponding to the buffer size used in previous experiments. Internal DRAM of SSD stores not only I/O data but also the address mapping table cache, and the address mapping table, in accordance with the address mapping scheme and map table caching policy of SSD [46]. Therefore, I/O data buffer size can be varied by the size of address mapping table and map cache; large address mapping table and map cache decrease the size of I/O data buffer. I/O data buffer size of 8-32 MB was used to observe the experimental results

TABLE 1. Description of workloads used in this study.

Workload Type	Workload Name	Read/ Write Ratio	I/O Size (GB)	Note
LSMKV	Rocks_load	30/70	4092.7	YCSB; Put operation 100%
LSMKV	Rocks_run	40/60	1882.9	YCSB; Update operation 50% Get operation 50%
Target app.	mds_0	30/70	10.6	MSR workloads
Target app.	hm_0	33/67	30.4	
Target app.	prxy_0	5/95	56.8	
Target app.	prn_0	22/78	59.0	
Target app.	usr_0	73/27	48.3	
Target app.	rsrch_0	11/89	12.2	
Target app.	wdev_0	28/72	9.8	
LSMKV+ Target app.	Rocks_load+ mds_0 (W1)	LSMKV: 30/70 App.: 5/95	4092.7 +0.7 =4093.4	Experimental workload of this paper
LSMKV+ Target app.	Rocks_load+ hm_0 (W2)	LSMKV: 30/70 App.: 31/69	4092.7 +2.3 =4095	
LSMKV+ Target app.	Rocks_load+ prxy_0 (W3)	LSMKV: 30/70 App.: 9/91	4092.7 +2.9 =4095.6	
LSMKV+ Target app.	Rocks_load+ prn_0 (W4)	LSMKV: 30/70 App.: 30/70	4092.7 +4.6 =4097.3	
LSMKV+ Target app.	Rocks_load+ usr_0 (W5)	LSMKV: 30/70 App.: 78/22	4092.7 +6.2 =4098.9	
LSMKV+ Target app.	Rocks_load+ rsrch_0 (W6)	LSMKV: 30/70 App.: 7/93	4092.7 +1.1 =4093.8	
LSMKV+ Target app.	Rocks_load+ wdev_0 (W7)	LSMKV: 30/70 App.: 31/69	4092.7 +1.0 =4093.7	
LSMKV+ Target app.	Rocks_run+ mds_0 (W8)	LSMKV: 40/60 App.: 30/70	1669.4 +10.6 =1680.0	
LSMKV+ Target app.	Rocks_run+ hm_0 (W9)	LSMKV: 40/60 App.: 33/67	1669.3 +30.4 =1699.7	
LSMKV+ Target app.	Rocks_run+ prxy_0 (W10)	LSMKV: 40/60 App.: 5/95	1664.9 +56.8 =1721.7	
LSMKV+ Target app.	Rocks_run+ prn_0 (W11)	LSMKV: 40/60 App.: 22/78	1669.3 +59.0 =1728.3	
LSMKV+ Target app.	Rocks_run+ usr_0 (W12)	LSMKV: 40/60 App.: 73/27	1669.3 +48.3 =1717.6	
LSMKV+ Target app.	Rocks_run+ rsrch_0 (W13)	LSMKV: 40/60 App.: 11/89	1669.3 +12.2 =1681.5	
LSMKV+ Target app.	Rocks_run+ wdev_0 (W14)	LSMKV: 40/60 App.: 28/72	1669.4 +9.8 =1679.2	

for the small I/O data buffer, and 64-128MB was used, vise versa. The assumed size of the SSD in the experiment was 300 GB and the page size was 4 KB. The LAT chunk size of the proposed method was 1 MB. The experiment for the proposed method was performed after ensuring that the size of the available data buffer was equal to the total buffer size with the LAT size removed ($available\_buffer\_size = total\_buffer\_size - LAT\_size$). According to equation (1), the size of the LAT used in the experiment, expressed in byte scale, was derived from equation (2). The result of the equation (2) is 75 KB. The SSD and chunk sizes affect the LAT size. As the SSD size increases, the size of the address space to be mapped by the LAT increases, thereby increasing the LAT size. If the chunk size is large, the LAT size is small because the entire address space of the SSD can be expressed with fewer indices. In the experiment, the proposed method classified sequential writes of 16 KB or more as large-sequential writes.

$$ByteSize_{LAT} = (300GB/1MB) \times ((1bit + 1bit)/8) \quad (2)$$

The parameter values of the existing algorithms used in the experiments were configured to the originally proposed values. For example, in a two-level LRU, the length of a hot list is 1/3rd of the total buffer size. The data size threshold that distinguishes sequential and random requests in a VBBMS is 4 KB. The virtual block sizes of SRSR and RRSR of VBBMS are 6 and 8 pages, respectively.

B. EXPERIMENTAL RESULTS

Figure 13 demonstrates the target application-specific buffer hit ratio ($HitRatio_{TA}$) of the existing buffer management (page replacement) policy and the proposed method. The x-axis represents the buffer size and buffer management policy and the y-axis represents $HitRatio_{TA}$. The graphs from (a) to (n) show the experimental results for workloads W1 to W14 listed in Table 1, respectively.

The storage access patterns of RocksDB_load and RocksDB_run are as follows. Both RocksDB_load and RocksDB_run have low buffer hit ratios. The buffer hit ratio of RocksDB_load was approximately 0.01–0.04% and that of RocksDB_run was approximately 0.2–0.8%. In RocksDB_load, I/O commands were requested more intensively than RocksDB_run; moreover, in RocksDB_run, I/O commands of a size smaller than RocksDB_load were frequently requested.

Figure 13 shows that the proposed method can increase $HitRatio_{TA}$ in comparison to the existing methods. Based on VBBMS, the most recent among the three algorithms that were compared, $HitRatio_{TA}$ of the proposed method ranges from a minimum of 0.56 times ((e) 16 MB in Fig. 13) to a maximum of 3.48 times ((l) 64 MB in Fig. 13).

As depicted in Fig. 13, the proposed method and VBBMS exhibit a high $HitRatio_{TA}$ for workloads (a)–(g) and (j), excluding (e). For workloads (h)–(n), the proposed method, PTLRU, and two-level LRU exhibit a high $HitRatio_{TA}$. The proposed method and VBBMS, which are size-based data separation techniques, classify large-sequential requests as cold requests. PTLRU and two-level LRU classify frequently hit data in the buffer as hot data. In workloads (a) to (g), RocksDB_load requests intensive cold sequential writes.

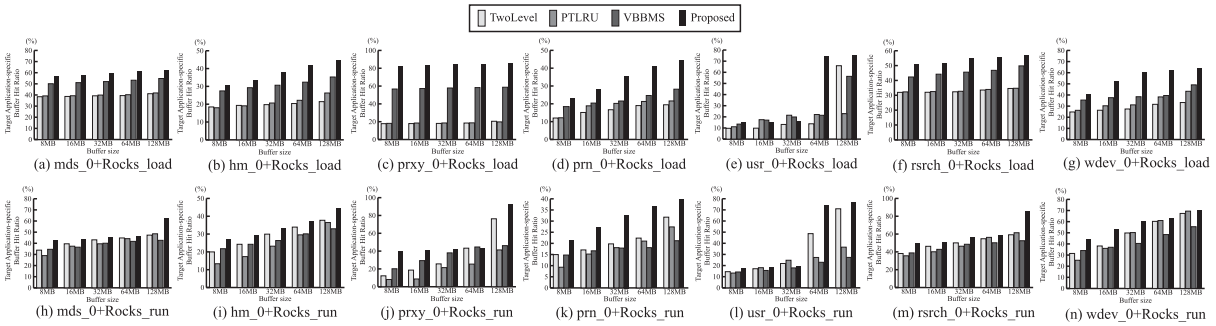


FIGURE 13. Buffer hit ratio of target application; (a) to (n) represent the experimental results from workloads W1 to W14, respectively.

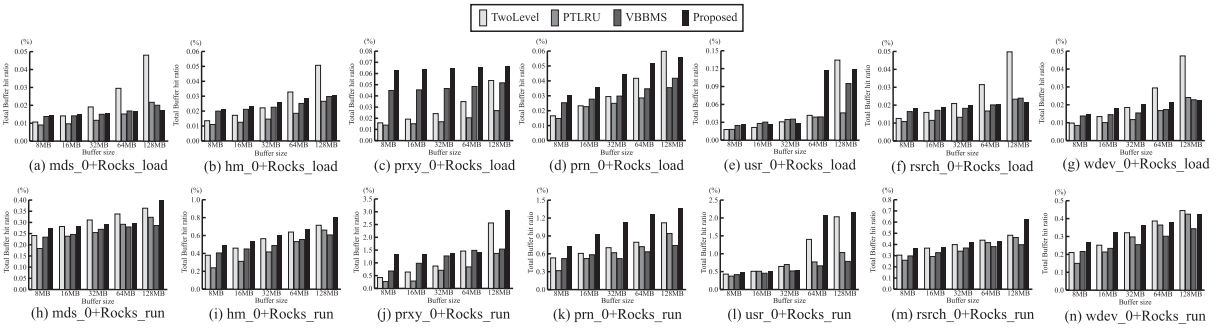


FIGURE 14. Total buffer hit ratio; (a) to (n) represent the experimental results from workloads W1 to W14, respectively.

Because size-based data separation stores the large-sequential cold data of RocksDB separately from other data, the proposed method and VBBMS effectively preserved the hot data of the target application in the buffer. In RocksDB_run, cold small-random and large-sequential reads were performed intensively and data I/O was requested less intensively than RocksDB_load. Therefore, for PTLRU and two-level LRU, the frequency of evicting hot data before being re-accessed was lower in workloads (h) to (n) in comparison to that in workloads (a) to (g). Moreover, in workloads (a) to (g), only a part of the workload of target application was used for the experiment, and in workloads (h) to (n), most of the workload of target application was used for the experiment. Therefore, even for the same target application, $HitRatio_{TA}$ in workloads (a) to (g) and (h) to (n) can differ.

In most cases, $HitRatio_{TA}$ of the proposed method was higher than that of the VBBMS. Because VBBMS allocates all small-random I/O to the hot region (RRSR), append-only data requested by small-random I/O can evict hot data in the hot region. In contrast, the proposed method divides small-random read into read for append-only data and read for non-append-only data. The I/O for append-only and non-append-only data is then allocated to different buffer spaces. The proposed method uses two small, fixed-length LRU lists to store append-only data. Therefore, even if the I/O of append-only data is intensively requested, the buffer of the proposed method only evicts a small amount of non-append-only data. The non-append-only data are considered more likely to be hit in the buffer than append-only data in our method.

In particular, as shown in (c) and (j) of Fig. 13, $HitRatio_{TA}$ of the proposed method was high. In the prxy_0 workload, data updates are frequently requested over a wide range of addresses. Furthermore, hot data candidates, which can be re-accessed in the near future, are intensively evicted before being re-accessed due to the append-only data I/O of LSMKV. Therefore, the proposed method, which allocates large-sequential append-only data to a small buffer space, can significantly increase $HitRatio_{TA}$.

However, for a buffer size of 8-32 MB of case (e) displayed in Fig. 13, $HitRatio_{TA}$ of the proposed method is lower than that of PTLRU and VBBMS. For usr_0, which is the target application of (e), buffer which size is 32 MB or less was insufficient to load most of hot data of the target application. Moreover data re-access is frequently requested via large-sequential writes and the average size of write re-access in usr_0 is approximately 25 KB. Size-based data separation predicts a large-sequential I/O as a cold request. Therefore, when the average request size of the data re-access command is large, $HitRatio_{TA}$ of the size-based technique is lower than that of the access-frequency-based technique. The proposed method allocates large-sequential hot writes to the SORB area where non-append-only data are stored; however, VBBMS allocates all large-sequential writes to the SRSR area where cold data are stored. Accordingly, for the buffer size of 16 MB of case (e) shown in Fig. 13, $WriteHitRatio_{TA}$

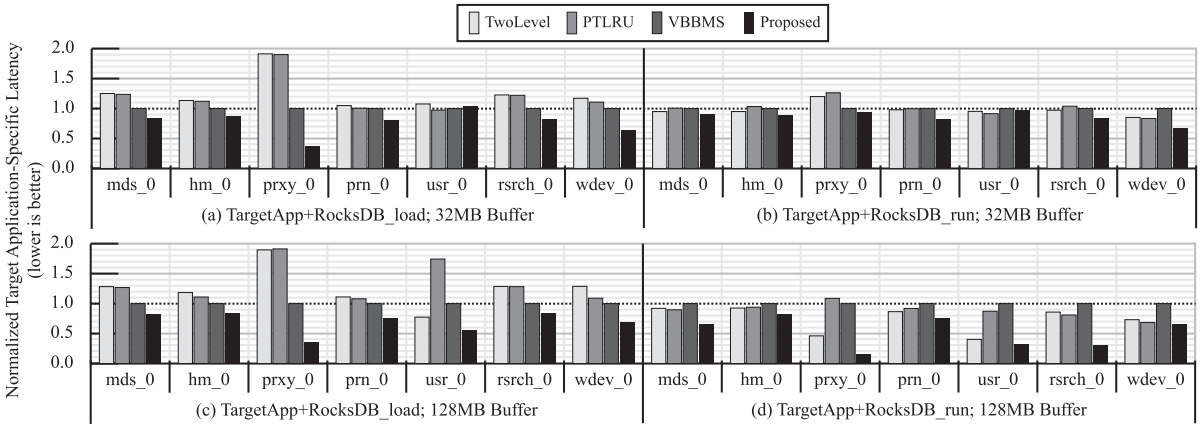


FIGURE 15. Storage access latency of target application in a 32 MB and 128 MB buffer. The lower label on the x-axis indicates the experimental workload and buffer size, while the upper label indicates the target application of the experimental workload.

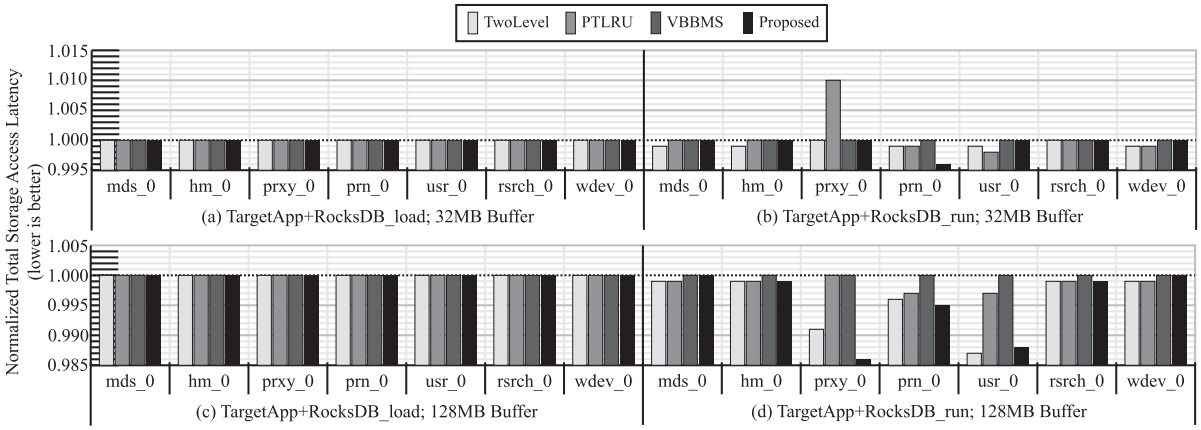


FIGURE 16. Total storage access latency in a 32 MB and 128 MB buffer. The lower label on the x-axis indicates the experimental workload and buffer size, while the upper label indicates the target application of the experimental workload.

($BufferHitRatio_{TA}$ for write request) of the proposed method was higher than that of VBBMS. In our additional experiments, $WriteHitRatio_{TA}$ of the proposed method was approximately 65.7%, whereas that of VBBMS was approximately 54%. Nevertheless, $HitRatio_{TA}$ of the proposed method was lower than that of VBBMS as shown in (e) 16 MB of Fig. 13, because (1) $ReadHitRatio_{TA}$ ($BufferHitRatio_{TA}$ for read request) of VBBMS is higher than that of the proposed method and (2) usr_0 is a read-intensive workload. Because VBBMS allocated a large-sequential request to the SRSR area, the hot data in the RRSR area were not evicted due to large-sequential I/O. Consequently, $ReadHitRatio_{TA}$ of the proposed method (approximately 1.1%) was lower than that of VBBMS (approximately 6.7%; approximately 0.3% and 6% in the SRSR and RRSR regions, respectively).

For the buffer size of 64-128 MB of case (e) and (l) in Fig. 13, experimental results show that the proposed method is more effective in securing the buffer size required for high $HitRatio_{TA}$; $HitRatio_{TA}$ of the proposed method was high because $ReadHitRatio_{TA}$ greatly increased (approximately 75%) as the available data buffer size increased. In the case of the 128 MB buffer, $HitRatio_{TA}$ of the proposed method, as well as those of two-level LRU and VBBMS were high, but $HitRatio_{TA}$ of the proposed method was the highest. In the proposed method, as the size of the data I/O buffer increases, the size of SORB increases whereas the buffer size for the append-only I/O is fixed. In the case of the two-level LRU and VBBMS, buffer space for hot data and cold data are both increased because the buffer spaces are determined by the ratio of the total buffer size. PTLRU configures LC, LD and LH as a linked list. The cold data of the LH list (the least recently used data of the LH list) are evicted according to the predefined probability. If the size of the LH list is large and the cold data of the LH list are not evicted, the size of the LC list becomes small and cold data with a high probability of re-access can be evicted before re-access.

As shown in case (l) of Fig. 13, $HitRatio_{TA}$ of the proposed method was greater than that of VBBMS and similar to that of PTLRU (8-16 MB) and two-level LRU (128 MB). $HitRatio_{TA}$ of the proposed method was higher in case (l) than

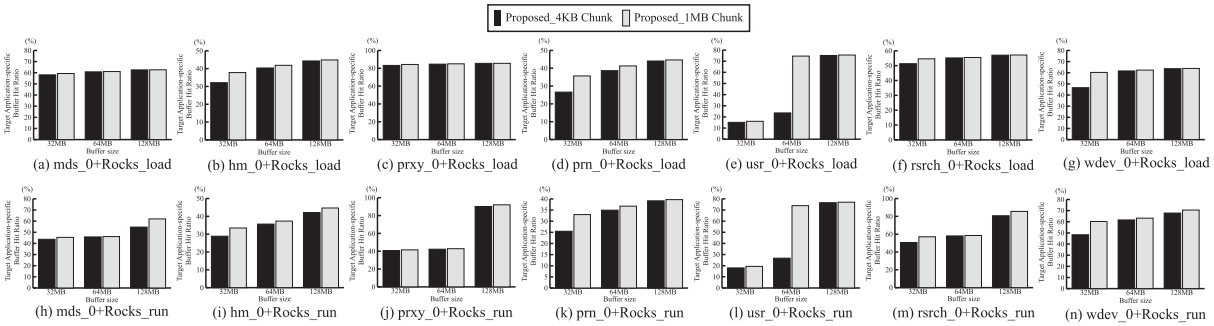


FIGURE 17. Buffer hit ratio of the target application of the proposed method for various chunk sizes, in case 32-128 MB buffer. (a) to (n) presents the experimental results for W1 to W14, respectively.

in case (e) even though these experiments were performed on the same target application. The proposed method stores small-random writes of RocksDB_load and large-sequential hot writes of usr_0 in the SORB area. In RocksDB_run, small-random write access, which is stored in the SORB area, is less frequent than in RocksDB_load. Therefore, in case (l), for the hot data of usr_0 stored in the SORB area, there were fewer evictions in comparison to case (e). When comparing (l) and (e), although $HitRatio_{TA}$ of PTLRU was similar, $HitRatio_{TA}$ of VBBMS decreased. Because VBBMS allocates all small-random I/O requests that are not hit in the buffer to the RRSR area, the data of usr_0 stored in the RRSR are frequently evicted. In case (e) 128 MB and (l) 128 MB, $HitRatio_{TA}$ of two-level LRU was high because the size of hot LRU was sufficiently large to retain the hot data of the target application, and the hot data candidates were not evicted prior to re-access because of the large candidate LRU.

The proposed method stores LAT in the buffer, resulting in a small available buffer size. However, since proposed method adopts coarse-grained mapping for LAT, LAT does not require a large memory space; LAT requires 75 KB for 1 MB chunk. Therefore, the proposed method can increase $HitRatio_{TA}$ in a small buffer. For example, when the buffer size was 8 MB in Fig. 13, the proposed method increased $HitRatio_{TA}$ by 74% than that of existing algorithms on average.

Figure 14 illustrates the total buffer hit ratio ($HitRatio_{Total}$) of the existing buffer management policy and the proposed method. Because the buffer hit ratio of RocksDB is low and the I/O scale of RocksDB is large in comparison to those of the target application, $HitRatio_{Total}$ in the experiment was lower than $HitRatio_{TA}$. In Fig. 14, $HitRatio_{Total}$ was approximately 0.008–3.04%.

The proposed method increased $HitRatio_{TA}$ without significantly reducing the *HitRatio* of RocksDB. Therefore, in many cases of the experiment, the proposed method improved the $HitRatio_{Total}$. However, the proposed method can reduce the buffer hit ratio for large-sequential writes. Furthermore, in the proposed method, data which are intermittently re-accessed are evicted from the SORB. Because the proposed method only allocates a small buffer space for large-sequential write requests and does not provide the separated buffer space for hot data. In Fig. 14 (a)-(g) 64-128 MB, except (c), two-level LRU shows higher $HitRatio_{Total}$ than the compared method, including the proposed method. As an example, in Fig. 14 (e) 128 MB, although $HitRatio_{TA}$ of the proposed method was 1.14 times that of two-level LRU, $HitRatio_{Total}$ of the proposed method was 0.89 times. Two-level LRU retained the small-sized hot data of LSMKV in hot data list while the several hot data candidates of target application were evicted from candidate data list. Whereas, in case of the proposed method, the small-sized hot data of LSMKV were evicted from SORB region because of data I/O of target application. Therefore, the proposed method shows higher $HitRatio_{TA}$ and lower $HitRatio_{Total}$ than that of two-level LRU. The major differences between the two-level LRU and proposed method are (1) two-level LRU has separated buffer space for hot data and (2) the proposed method has larger buffer space (SORB) for hot data candidates.

Even though the proposed method reduces $HitRatio_{Total}$, the storage access performance of the system is maintained. This is because $HitRatio_{Total}$ is low in systems in which the LSMKV operates, and the buffer does not significantly reduce the storage access latency of the LSMKV. According to Fig. 16, the total storage access latency of the method proposed in (e) was 1.002 times that of VBBMS. The proposed method was primarily employed to reduce the storage access latency of the target application in an environment where LSMKV and the target application access the storage simultaneously without significantly sacrificing the overall storage access performance. As shown in Fig. 16, total storage access latency of the proposed method was 0.986-1.002 times that of VBBMS.

Figure 15 demonstrates the storage access latency of the target application ($Latency_{TA}$) of the existing buffer management and proposed methods in the 32 MB and 128 MB buffers; the y-axis represents $Latency_{TA}$ and x-axis represents the specific workload, buffer size and the applied buffer management method. The value indicated in each column bar was normalized based on $Latency_{TA}$ of VBBMS, which is the

most recent among the algorithms compared in this study. The lower the storage access latency, the shorter the waiting time for I/O request completion, leading to a faster I/O response speed.

Because the proposed method reads the entry of the LAT of the requested logical address for each storage access, it incurs additional memory access and LAT search overhead. Herein, the entry of LAT can be randomly accessed because LAT is an array which is indexed by LCA. The search for the entry of LAT with a given address can be processed within the constant time of Big-O notation. Therefore, the latency of additional memory accesses for LAT is included in the storage access latency measurement of the proposed method but the search time for LAT is not included.

As shown in Fig. 15 and Fig. 13, the higher the $HitRatio_{TA}$ of the algorithm, the lower the $Latency_{TA}$. However, the performance gaps between the algorithms in the two figures are different. Because the access latency of NAND flash is significantly longer than that of DRAM [47], the storage access latency is more affected by the buffer miss ratio, i.e., the NAND access count, than the buffer hit ratio.

Figure 15 demonstrates that $Latency_{TA}$ of the proposed method was considerably low in comparison to that of other methods; especially for prxy_0 of Fig. 15 and usr_0 of Fig. 15 (c), (d) and rsrch_0 of Fig. 15 (d). This is because $HitRatio_{TA}$ of the proposed method was substantially high in aforementioned cases (approximately 76%-91%). Meanwhile, Fig. 13 demonstrates that $MissRatio_{TA}$ $(1 - HitRatio_{TA})$ of the proposed method was approximately 0.56-0.15 times that of VBBMS, which is similar to the difference in $Latency_{TA}$ between the two methods demonstrated in Fig. 15.

Figure 17 illustrates $HitRatio_{TA}$ for various chunk sizes of the proposed method in case 32-128 MB buffer. In the experiment, chunk sizes of 4 KB (same with the page size) and 1 MB (default) were used. The larger the chunk size of the proposed method, the smaller the LAT size. The size of LAT with 4 KB chunk was 18.75 MB, thus buffer less than 16 MB was not available. Consequently, the available buffer space of the proposed method increased, thereby increasing the $HitRatio_{TA}$.

However, an increase in chunk size may decrease $HitRatio_{TA}$. The proposed method manages the logical address space by dividing it into chunk units. If the chunk size is large, the likelihood of requesting I/O commands with different access patterns in the same chunk is high. For example, when random and sequential I/O are mixed in the narrow address space of a workload, the address space for both random and sequential I/O can be included in one chunk if a coarse-grained chunk is used. In this case, $HitRatio_{TA}$ can be reduced by increasing the chunk size.

VI. CONCLUSION

We presented an SSD internal buffer space separation and management method for environments where applications such as the LSMKV, which intensively requests data with low re-access frequency, access the storage concurrently with other applications. LSMKV intensively generates append-only data that are rarely re-accessed. The proposed method analyzed the storage access pattern for the append-only data of the LSMKV and detected I/O for the append-only data. Append-only data I/O can be detected based on (1) whether the incoming write access pattern matches the write pattern of append-only data or (2) if the read command is requested for the append-only data. To detect append-only data I/O, a table called LAT, which maps logical addresses, writes the size, and re-accesses the status of corresponding logical addresses, was proposed. As append-only data are considered to be cold data in the proposed method, the append-only data I/O was allocated to a separate buffer space, BAW, and BAR. Other non-append-only data I/O requests can be allocated to the entire buffer space; i.e., the buffer space where general I/O requests are allocated is SORB. When a buffer entry is to be evicted, the proposed method first evicts BAW and BAR. Thus, the storage access speed of applications, such as the LSMKV, that access the storage simultaneously is increased by alleviating the buffer eviction of data with a high probability of being re-accessed owing to data I/O in the LSMKV with a low buffer hit ratio. In comparison to the results of the existing buffer management schemes, our experimental results demonstrated that the proposed method can reduce the storage access latency of target applications by 30.84%, on average, when multiple applications, including the LSMKV, access the storage concurrently.

REFERENCES

[1] M. Bjørling, "From open-channel SSDs to zoned namespaces," in *Linux Storage Filesyst. Conf.*, Boston, MA, USA, 2019, pp. 1–18.
[2] C. C. Aggarwal, N. Ashish, and A. Sheth, "The Internet of Things: A survey from the data-centric perspective," in *Managing and Mining Sensor Data*. Springer, 2013, pp. 383–428.
[3] H. Jiang, F. Shen, S. Chen, K.-C. Li, and Y.-S. Jeong, "A secure and scalable storage system for aggregate data in IoT," *Future Gener. Comput. Syst.*, vol. 49, pp. 133–141, Aug. 2015.
[4] S. Ghemawat and J. Dean. *LevelDB*. Accessed: Mar. 10, 2021. [Online]. Available: https://github.com/google/leveldb
[5] D. Borthakur. *Rocksdb A Persistent Key-Value Store*. Accessed: Mar. 10, 2021. [Online]. Available: https://rocksdb.org/
[6] 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.
[7] M. N. Vora, "Hadoop-HBase for large-scale data," in *Proc. ICCSNT*, vol. 1, 2011, pp. 601–605.
[8] S. Dong, M. Callaghan, L. Galanis, D. Borthakur, T. Savor, and M. Strum, "Optimizing space amplification in rocksdb," in *Proc. CIDR*, vol. 3, 2017, p. 3.
[9] H. Sun, S. Dai, and J. Huang, "Cascaded write amplification of LSM-tree-based key-value stores underlying solid-state disks," *Microprocessors Microsyst.*, vol. 78, Oct. 2020, Art. no. 103217.
[10] L.-P. Chang and T.-W. Kuo, "An adaptive striping architecture for flash memory storage systems of embedded systems," in *Proc. 8th Real-Time Embedded Technol. Appl. Symp.*, 2002, pp. 187–196.
[11] C. Du, Y. Yao, J. Zhou, and X. Xu, "VBBMS: A novel buffer management strategy for NAND flash storage devices," *IEEE Trans. Consum. Electron.*, vol. 65, no. 2, pp. 134–141, May 2019.

[12] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "LRU-WSR: Integration of LRU and writes sequence reordering for flash memory," *IEEE Trans. Consum. Electron.*, vol. 54, no. 3, pp. 1215–1223, Aug. 2008.

[13] Z. Li, P. Jin, X. Su, K. Cui, and L. Yue, "CCF-LRU: A new buffer replacement algorithm for flash memory," *IEEE Trans. Consum. Electron.*, vol. 55, no. 3, pp. 1351–1359, Aug. 2009.

[14] P. Jin, Y. Ou, T. Härder, and Z. Li, "AD-LRU: An efficient buffer replacement algorithm for flash-based databases," *Data Knowl. Eng.*, vol. 72, pp. 83–102, Feb. 2012.

[15] J. Cui, W. Wu, Y. Wang, and Z. Duan, "PT-LRU: A probabilistic page replacement algorithm for NAND flash-based consumer electronics," *IEEE Trans. Consum. Electron.*, vol. 60, no. 4, pp. 614–622, Nov. 2014.

[16] Y. Yuan, Y. Shen, W. Li, D. Yu, L. Yan, and Y. Wang, "PR-LRU: A novel buffer replacement algorithm based on the probability of reference for flash memory," *IEEE Access*, vol. 5, pp. 12626–12634, 2017.

[17] Y. Yuan, J. Zhang, G. Han, G. Jia, L. Yan, and W. Li, "DPW-LRU: An efficient buffer management policy based on dynamic page weight for flash memory in cyber-physical systems," *IEEE Access*, vol. 7, pp. 58810–58821, 2019.

[18] A. Cassandra. (2014). *Apache Cassandra*. [Online]. Available: http://planetcassandra. org/what-is-apache-cassandra

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

[20] 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.

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

[22] J.-U. Kang, J.-S. Kim, C. Park, H. Park, and J. Lee, "A multi-channel architecture for high-performance NAND flash-based storage system," *J. Syst. Archit.*, vol. 53, no. 9, pp. 644–658, Sep. 2007.

[23] Y. Deng and J. Zhou, "Architectures and optimization methods of flash memory based storage systems," *J. Syst. Archit.*, vol. 57, no. 2, pp. 214–227, Feb. 2011.

[24] Y. Hu and X. Dong, "A kind of FTL scheme which keeps the high performance and lowers the capacity of RAM occupied by mapping table," in *Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS)*, Aug. 2016, pp. 1–2.

[25] C. Gao, L. Shi, K. Liu, C. J. Xue, J. Yang, and Y. Zhang, "Boosting the performance of SSDs via fully exploiting the plane level parallelism," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 9, pp. 2185–2200, Sep. 2020.

[26] S. Im and D. Shin, "Flash-aware RAID techniques for dependable and high-performance flash memory SSD," *IEEE Trans. Comput.*, vol. 60, no. 1, pp. 80–92, Jan. 2011.

[27] J. Kim, H. Kim, S. Lee, and Y. Won, "FTL design for TRIM command," in *Proc. 5th Int. Workshop Softw. Support Portable Storage*, 2010, pp. 7–12.

[28] D. Shin, "About SSD," in *Proc. Linux Storage Filesyst. Workshop*, 2008, pp. 1–27.

[29] T. Frankie, G. Hughes, and K. Kreutz-Delgado, "SSD trim commands considerably improve overprovisioning," in *Proc. Flash Memory Summit*, 2011, pp. 1–19.

[30] F. Geier, "The differences between SSD and HDD technology regarding forensic investigations," Tech. Rep., 2015.

[31] M. Jung and M. Kandemir, "Revisiting widely held SSD expectations and rethinking system-level implications," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 41, no. 1, pp. 203–216, Jun. 2013.

[32] Y.-S. Yoo, H. Lee, Y. Ryu, and H. Bahn, "Page replacement algorithms for NAND flash memory storages," in *Proc. Int. Conf. Comput. Sci. Appl.* Springer, 2007, pp. 201–212.

[33] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst.*, 2006, pp. 234–241.

[34] J. Lee and J.-S. Kim, "An empirical study of hot/cold data separation policies in solid state drives (SSDs)," in *Proc. 6th Int. Syst. Storage Conf.*, 2013, pp. 1–6.

[35] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, "FAB: Flash-aware buffer management policy for portable media players," *IEEE Trans. Consum. Electron.*, vol. 52, no. 2, pp. 485–493, May 2006.

[36] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," in *Proc. FAST*, vol. 8, 2008, pp. 1–14.

[37] J. Kwak, J. Lee, D. Lee, J. Jeong, G. Lee, J. Choi, and Y. H. Song, "GALRU: A group-aware buffer management scheme for flash storage systems," *IEEE Access*, vol. 8, pp. 185360–185372, 2020.

[38] 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.

[39] 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.

[40] 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)*, Feb. 2017, pp. 777–781.

[41] 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)*, May 2017, pp. 1–13.

[42] 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. Annu. Tech. Conf.* , 2015, pp. 71–82.

[43] 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.

[44] 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.

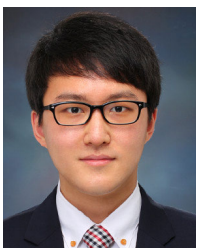
[45] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Trans. Storage*, vol. 4, no. 3, pp. 1–23, Nov. 2008.

[46] C. Gao, Y. Di, A. Deng, D. Liu, C. Ji, C. J. Xue, and L. Shi, "F2FS aware mapping cache design on solid state drives," in *Proc. IEEE 7th Non-Volatile Memory Syst. Appl. Symp. (NVMSA)*, Dec. 2018, pp. 31–36.

[47] C. Zambelli, G. Navarro, V. Sousa, I. L. Prejbeanu, and L. Perniola, "Phase change and magnetic memories for solid-state drive applications," *Proc. IEEE*, vol. 105, no. 9, pp. 1790–1811, Sep. 2017.

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

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



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

His research interests include embedded computing and NAND flash memories.

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

His research interests include embedded computing and IoT devices.

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 worked with the IBM T.J. Watson Research Center as a Research Staff Member, from 2015 to 2019. He is currently an Assistant Professor with Hanyang University, South Korea. His research interest includes the 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 academic activities, such as the Technical Program Committee of DATE 2018–2020 (the Co-Chair) and DAC 2018–2020, and the Technical Committee (DiSPS) in IEEE Signal Processing Society.

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

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

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