IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received September 21, 2020, accepted October 7, 2020, date of publication October 12, 2020, date of current version October 21, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3030089

GALRU: A Group-Aware Buffer Management Scheme for Flash Storage Systems

JAEWOOK KWAK[1], JUNGKEOL LEE[1], DAEYONG LEE[1], JOONYONG JEONG[1], GYEONGYONG 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 Ministry of Trade, Industry and Energy/Korea Institute for Industrial Economics and Trade (MOTIE/KEIT) through the Research and Development Program (Developing Processor–Memory–Storage Integrated Architecture for Low-Power, High-Performance Big Data Servers) under Grant 10077609.

ABSTRACT Many flash storage systems divide input/output (I/O) requests that require large amounts of data into sub-requests to exploit their internal parallelism. In this case, an I/O request can be completed only after all sub-requests have been completed. Thus, non-critical sub-requests that are completed quickly do not affect I/O latency. To efficiently reduce I/O latency, we propose a buffer management scheme that allocates buffer space by considering the relationship between the processing time of the sub-request and I/O latency. The proposed scheme prevents non-critical sub-requests from wasting ready-to-use buffer space by avoiding the situation in which buffer spaces that are and are not ready to use are allocated to an I/O request. To allocate the same type of buffer space to an I/O request, the proposed scheme first groups sub-requests derived from the same I/O request and then applies a policy for allocating buffer space in units of sub-request groups. When the ready-to-use buffer space is insufficient to be allocated to the sub-request group being processed at a given time, the proposed scheme does not allocate it to the sub-request group but it instead sets it aside for future I/O requests. The results of the experiments to test the proposed scheme show that it can reduce I/O latency by up to 24% compared with prevalent buffer management schemes.

INDEX TERMS Buffer management, flash memory, flash translation layer, flash storage system.

I. INTRODUCTION

To compensate for the difference in throughput between a flash memory device and the host interface, many flash storage systems use parallelism. The throughput of flash memory devices is substantially lower than that of the host interfaces [1]–[3]; consequently, flash memory devices can limit the throughput of the host interfaces. To overcome this limitation, the hardware architectures of many flash storage systems have been designed to utilize internal parallelism [4]–[6] at the channel, package, die, and plane levels. Many flash storage systems reconstruct I/O requests to use internal parallelism because the size of the data requested through I/O requests is not constant. An I/O request requesting a large amount of data is divided into sub-requests, depending on the unit of I/O operation of the flash memory device, and these sub-requests are processed in parallel.

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

Several studies have reported improvements in the I/O performance of flash storage systems by adjusting the processing times of the sub-requests. Because an I/O request divided into sub-requests can be completed only when all sub-requests have been completed, I/O latency is determined by the sub-request that has the longest processing time. If the processing of critical sub-requests that determine I/O latency can be accelerated by delaying the processing of non-critical sub-requests that do not affect I/O latency, the I/O performance of a flash storage system can be improved. Many studies have reported improvements in I/O schedulers [7], [8] and data allocation schemes [9]–[11] by adjusting the processing times of the sub-requests.

Because dirty data significantly affect the processing time of a sub-request, the buffer space that contains clean data must be efficiently managed to improve the I/O performance of the flash storage system. To prevent the speed of the host interface from being limited by the slow speed of the flash interface [12]–[14], a flash storage system uses

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

a high-performance memory device as a buffer to temporarily store the data transferred between the host system and the flash memory device. The data accessed in the past remain in the buffer, and thus the sub-request can use the buffer space after performing an eviction process that allocates the buffer space and removes data already present in it. The clean data can be removed from the buffer immediately; however, dirty data can be removed only after being written to the flash memory device. Because the flash memory device takes a long time for the write operation, the dirtiness of the data present in the allocated buffer space significantly influences the processing time of the sub-request. If the buffer space containing clean data is used for critical sub-requests, I/O latency can be reduced. It is necessary to prevent the buffer space containing clean data from being wasted by non-critical sub-requests because the amount of clean data in the buffer is not always sufficient.

However, many buffer management schemes [15]–[28] have limitations in efficiently managing the buffer space containing clean data because the size of the unit in which the eviction process is executed is not the same as the size of the data requested by the I/O request. Buffer management schemes for flash storage systems manage the buffer space by dividing it into buffer entries, which are units of the same size as a page or a block. The value of each entry is determined by examining such characteristics of the data as access frequency and dirtiness, but the criteria for determining this value differ depending on the buffer management scheme. If there are no free buffer entries, the entry with the lowest value is usually allocated to a new sub-request. Because the eviction process is performed in a unit of the buffer entry whenever the free buffer entry is insufficient for a given sub-request, clean and dirty data can be stored together in buffer entries allocated for an I/O request that is the parent of the sub-requests. In this case, a sub-request assigned with a dirty entry, which is the buffer entry containing dirty data, is likely to be a critical sub-request because the flash device takes a long time for the write operation. On the contrary, a sub-request assigned with a clean entry, which is the buffer entry containing clean data, is likely to be a non-critical sub-request. As a result, clean entries can be wasted by non-critical sub-requests.

We propose a buffer management scheme called the group-aware least recently used (GALRU) scheme to efficiently manage clean entries. The basic idea of the GALRU is to unify the type of buffer entries allocated for I/O requests as clean or dirty entries. Allocating clean and dirty entries together to an I/O request is not effective for reducing its processing latency, as mentioned above. In addition, it can reduce the probability that the latency of future I/O requests can be reduced because it reduces the number of the clean entries that can be used for future I/O requests. The GALRU increases opportunities for clean entries to be used to reduce I/O latency by avoiding the allocation of clean and dirty entries together to an I/O request. It has the following features:

- The GALRU groups sub-requests derived from the same I/O request and applies an eviction policy in units of a sub-request group. The eviction policies can limit the type of buffer entries to be allocated to sub-requests. Furthermore, the GALRU applies the same eviction policy to sub-requests belonging to the same group; thus, such sub-requests can use the same type of buffer entries.
- The eviction policy is determined depending on the number of clean entries. If the number of clean entries is insufficient to be allocated to the sub-request group being processed, the GALRU uses an eviction policy that selects a dirty entry as the victim entry so that clean entries can be accumulated in the buffer to reduce the I/O latency of future I/O requests. If the number of clean entries is sufficient, the GALRU uses an eviction policy that selects a clean entry as the victim entry to reduce the latency of the sub-request group being processed.
- Even though the GALRU applies an eviction policy in units of a sub-request group, it executes the eviction process in a unit of the buffer entry. Because the unit of the eviction process is identical to that of many currently available buffer management schemes, the eviction algorithms of the latter can be improved by adopting the idea of the GALRU without requiring major changes. Section 5 presents case studies of the application of the eviction policies of the GALRU to buffer management schemes.

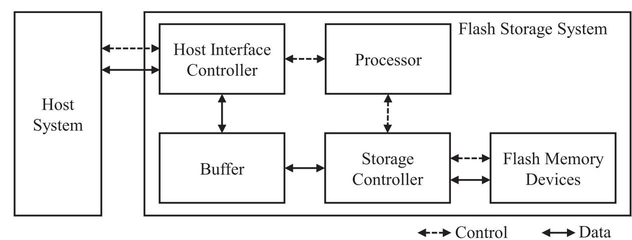


FIGURE 1. Architecture of flash storage system.

II. BACKGROUND AND RELATED WORK

A. FLASH STORAGE SYSTEM ARCHITECTURE

Figure 1 shows the common hardware components of a flash storage system [29]–[33]. The processor, which executes the algorithm for the flash translation layer and the buffer, generates control words for the host interface controller and the storage controller to process sub-requests. The host interface controller, which manages communication between the host system and the flash storage system, moves data requested by the sub-request between the host system and the buffer. The storage controller, which controls the flash memory devices, moves the data requested by the sub-request between the flash memory device and the buffer. Many flash storage system contain flash memory devices constructed using a multi-channel, multi-way architecture [34], [35] to utilize channel-level and die-level parallelism.

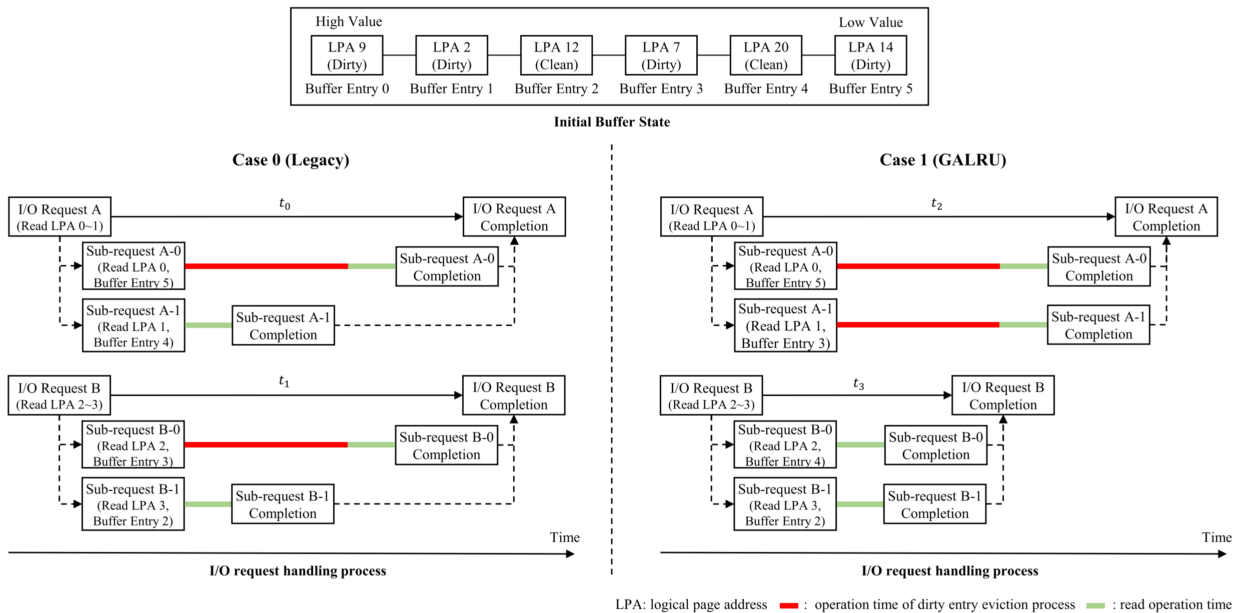


FIGURE 2. Comparison of I/O request handling processes of legacy schemes and GALRU.

B. RELATED WORK

The buffer management scheme has a significant influence on the performance of the flash storage system because it can modulate the pattern of access to the flash device. In general, the buffer management scheme improves the performance of the flash storage system by storing frequently accessed data in the buffer for a long time or changing the pattern of write access of the host system to one suitable for reducing the write amplification factor. The buffer structure of prevalent buffer management schemes can be classified into three types: a write-only structure in which only the write buffer exists, a separate structure in which the read and write buffers are separated from each other, and an integrated structure in which the read and write buffers share the buffer space. The write-only structure [36]–[39] cannot reduce the frequency of read access to the flash device. The separate structure [40]–[42] struggles to cope with changes in workload because it is difficult to dynamically adjust the sizes of the read and write buffers. In addition, when read and write requests access the same address, it difficult to efficiently manage the buffer space compared with that in the integrated structure.

Buffer management schemes for integrated structures [15]–[28], [43] use the dirtiness of the data as a criterion for selecting the victim entry for the eviction process because the flash device has a write life limit and takes a long time for the write operation. To reduce the frequency of flash write operations, the CFLRU [15] contained an eviction policy based on the clean-first method, which involves selecting a clean entry over a dirty entry as the victim entry of the eviction process. Many buffer management schemes [16]–[28] have since featured eviction policies based on the clean-first method. To improve the buffer hit ratio, they mainly use an eviction policy that integrates the clean-first method with a cold-first method. The cold-first method selects a cold entry over a dirty entry as the victim entry of the eviction process; the cold entry contains data that are unlikely to be accessed again, and the dirty entry contains data that are likely to be accessed again. Even though the criteria for determining the value differ depending on the buffer management scheme, the schemes usually assign low values to a cold entry or a clean entry, and select the one with the lowest value as the victim entry of the eviction process. They usually employ page-level buffer entries because the accuracy of classifying the data improves as the granularity of management decreases.

III. MOTIVATION

Buffer management schemes for integrated structures are limited in the extent to which they can improve the performance of the flash storage system due to the eviction process in units of buffer entry. Case 0 in Figure 2 shows the problem encountered by buffer management schemes for integrated structures. As mentioned above, they allocate the buffer entry with the lowest value to a new sub-request. If clean and dirty entries are mixed among low-value buffer entries, clean and dirty entries can be allocated together for an I/O request as shown in Case 0. I/O operations for the sub-requests A-0 and B-0, which use dirty entries, are delayed by the eviction process for writing dirty data to the flash device, whereas I/O operations for the sub-requests A-1 and B-1, which use clean entries, are completed early. As a result, I/O latency (t_0 and t_1) is determined by the sub-requests A-0 and B-0 using the dirty entry, and clean entries do not contribute to reducing I/O latency.

The GALRU is proposed to efficiently manage clean entries to reduce I/O latency. It can create an opportunity for a clean entry, which was previously used by a non-critical sub-request of a preceding I/O request, to be used for a critical sub-request of a subsequent I/O request. Case 1 of Figure 2 shows the basic idea of the GALRU. It allocates buffer entry 3 to sub-request A-1, instead of buffer entry 4, and allocates buffer entry 4 to sub-request B-0 instead of buffer entry 3, to unify the types of buffer entries allocated for I/O requests. As a result, even though the I/O latencies of I/O request A in Case 0 (t_0) and Case 1 (t_2) are the same, the I/O latency (t_3) of I/O request B in Case 1 is less than the I/O latency (t_1) of I/O request B in Case 0.

IV. PROPOSED GALRU SCHEME

The GALRU is a buffer management scheme that applies eviction policies in units of a sub-request (SR) group. The SR group contains sub-requests derived from the same I/O request. The GALRU processes an SR group by selecting one of three eviction policies. The eviction policy of the GALRU determines the victim entry of the eviction process depending on the dirtiness of the buffer entry. Because sub-requests belonging to the same group are processed by the same eviction policy, the dirtiness of buffer entries allocated for I/O requests are unified. Details of the GALRU are described in the sub-sections below.

A. BUFFER STRUCTURE

Figure 3 shows the buffer structure of the GALRU. It manages the buffer by dividing it into a common region and a victim region. The common region is used to manage recently accessed data, and the victim region is used for managing data pushed out of the common region. Data in the buffer are managed in units of the page-level buffer entry. The GALRU maintains the sizes of the common region and victim region.

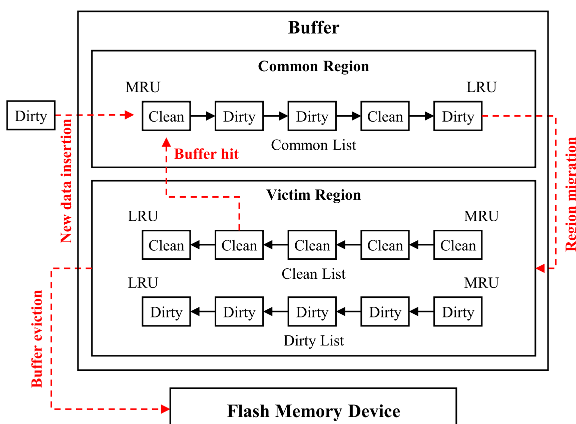


FIGURE 3. Buffer structure of the GALRU.

The buffer entries of the common region are managed by a common list that sorts them using the LRU algorithm. The buffer entry for data accessed by a new sub-request is moved to the most recently used (MRU) location of the common list. If the buffer entry is input from outside the common region, the entry of the LRU location of the common list is moved to the victim region to maintain the size of the common region. This process is called region migration.

The victim region consists of two lists that manage different types of buffer entries. The dirty list, which is for dirty entries, and the clean list, which is for clean entries, use the LRU algorithm to sort their buffer entries. The buffer entry moved by region migration is entered into one of the two lists depending on its dirtiness. The victim entry of the eviction process is selected from among buffer entries in the victim region and moved to the common region after being used by a new sub-request.

B. EVICTION POLICY

Before performing the eviction process for an SR group, the GALRU selects one of the following three policies: clean only (CO), dirty only (DO), or mix available (MA). The CO policy selects the LRU buffer entry of the clean list as victim entry, the DO policy selects the LRU buffer entry of the dirty list as victim entry, and the MA policy selects the oldest among the LRU buffer entries of each list as victim entry. Because the GALRU applies the same eviction policy to sub-requests belonging to the same SR group, the dirtiness of buffer entries allocated to an I/O request is unified except when the MA policy is applied.

The GALRU selects an eviction policy by comparing the size of an SR group with the number of clean entries in the victim region. The size of an SR group refers to the sum of data sizes requested by sub-requests of the group. In other words, the size of an SR group is identical to the size of data requested by the I/O request. To unify the dirtiness of buffer entries allocated to an SR group, the number of clean or dirty entries in the victim region should be greater than the size of the SR group. The GALRU uses the CO policy only if the size of the clean list is larger than that of the given SR group. Likewise, it uses the DO policy only if the size of the dirty list is larger than that of the given SR group. Because the eviction process of a clean entry is faster than that of a dirty entry, the GALRU uses the CO policy when the size of each list is larger than that of the given SR group. Furthermore, it uses the MA policy only if the size of each list is smaller than that of the given SR group.

C. SR GROUP DISTINCTION

The GALRU uses an identifier (RID) for the I/O request to classify sub-requests belonging to different SR groups. Numerous flash storage systems insert an RID into a sub-request to complete the I/O request. The flash storage system can determine the operating status of the I/O request by checking the RID of the completed sub-requests. Because an RID contains unique information pertaining to an I/O request, it is useful for distinguishing SR groups. Figure 4 shows an example in which the GALRU distinguishes among SR groups. It determines that sub-requests containing the same RID belong to the same SR group.

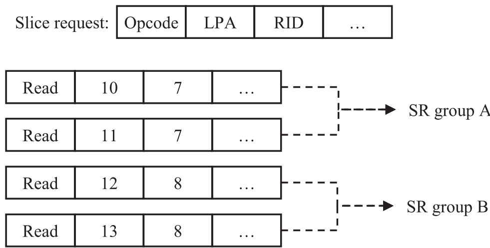


FIGURE 4. Example of SR group distinction.

D. EVICTION POLICY REGISTER

To apply the same eviction policy to sub-requests belonging to the same SR group, the GALRU uses an eviction policy register (EPR) to maintain information regarding the eviction process. The EPR stores the RID and eviction policy identifier (PID) of the SR group being processed. Furthermore, the PID can be set to one of four states: *reset*, CO, DO, and MA. *Reset* indicates that the eviction policy has not been determined yet. When the GALRU starts processing a new SR group, it updates the RID of the EPR with the RID of the new SR group and sets the PID of the EPR to *reset*. It selects a new eviction policy only when the PID of the EPR is *reset*. Because the PID of the EPR is retained until a new SR group is processed, the GALRU can apply the same eviction policy to sub-requests belonging to the same SR group by checking the PID of the EPR.

E. GALRU ALGORITHM

Algorithm 1 shows the pseudocode of the sub-request handling process. This process is performed in two stages. In the first stage, the GALRU compares the RID of the sub-request with that of the EPR to determine whether the request belongs to an SR group being processed (lines 1–3). If the RIDs are the same, the GALRU retains the information of the EPR. If the RIDs are not the same, it updates the information of the EPR for the new SR group (line 2). In the second step, the GALRU reorganizes the buffer following several steps that are determined depending on the location of the data requested by the sub-request. If the requested data are in the common region, the GALRU performs a list sort, where this moves the buffer entry storing the requested data to the MRU location of the common list (line 6). If the requested data are in the victim region, the GALRU performs a list sort and region migration, where this keeps the size of each region constant (lines 8–13). If the requested data do not exist in the buffer, the GALRU performs a list sort, region migration, and the eviction process, which allocates buffer entry for the requested data (lines 16–28).

Algorithm 2 shows the process of victim selection. This process is performed in two stages. In the first stage, the GALRU checks the PID of the EPR to determine whether a new eviction policy should be chosen (lines 1–9). It selects a new eviction policy only if the PID is *reset*. If the PID is not *reset*, the GALRU keeps the PID to use the eviction policy chosen for previous sub-requests. In the second stage, it selects the victim entry according to the eviction policy determined in the first stage (lines 10–21).

Algorithm 1 Sub-Request Handling Process

CR; common region; VR: victim region; CML: common list of CR; CL: clean list of VR; DL: dirty list of VR; EPR: eviction policy register; LPA: logical page address; D(X): data of X; E(X): buffer entry of X; RID: ID of I/O request; *victim*: victim entry

Input: LPA, RID

Output: Reference of E(D(LPA))

```
1:  if RID != RID of EPR then
2:      Reset EPR and update RID of EPR;
3:  end if
4:  if D(LPA) ∈ buffer then
5:      if D(LPA) ∈ CR then
6:          Move E(D(LPA)) to the MRU of CML;
7:      else
8:          Move E(D(LPA)) to the MRU of CML;
9:          if E(LRU of CML) are the clean data then
10:             Move E(LRU of CML) to the MRU of CL;
11:         else
12:             Move E(LRU of CML) to the MRU of DL;
13:         end if
14:     end if
15: else
16:     victim ← Victim_Selection()
17:     if D(victim) are the dirty data then
18:         Evict D(victim) to the flash memory device;
19:     else
20:         Discard D(victim);
21:     end if
22:     Store D(LPA) to victim
23:     Insert E(D(LPA)) into the MRU of CML;
24:     if E(LRU of CML) are the clean data then
25:         Move E(LRU of CML) to the MRU of CL;
26:     else
27:         Move E(LRU of CML) to the MRU of DL;
28:     end if
29: end if
30: return E(D(LPA));
```

F. COMPLEXITY OF GALRU ALGORITHM

Because the GALRU uses an algorithm based on the LRU algorithm, this section analyzes the additional space and time complexities that the GALRU algorithm incurs compared with the LRU algorithm. The additional space complexity of the GALRU is generated by the EPR and additional LRU lists. Compared with the LRU algorithm, metadata for the two LRU lists is added because the GALRU uses three LRU lists. Because the sizes of the metadata for the EPR and LRU lists are constant, the additional space complexity is O(1). The additional time complexity of the GALRU is incurred

Algorithm 2 Victim Selection

CR; common region; VR: victim region; CL: clean list of VR; DL: dirty list of VR; EPR: eviction policy register; CO: clean-only eviction policy; DO: dirty-only eviction policy; MA: mix-available eviction policy; E(X): buffer entry of X; RID: ID of I/O request; *victim*: victim entry

Input: RID

Output: Reference of *victim*

```
1:  if PID of EPR == reset then
2:      if Size of the SR group of RID ≤ size of CL then
3:          Set PID of EPR to CO;
4:      else if Size of the SR group of RID ≤ size of DL then
5:          Set PID of EPR to DO;
6:      else
7:          Set PID of EPR to MA;
8:      end if
9:  end if
10: if PID of EPR == CO then
11:     victim ← E(LRU of CL);
12: else if PID of EPR == DO then
13:     victim ← E(LRU of DL);
14: else
15:     if E(LRU of CL) are older than E(LRU of DL) then
16:         victim ← E(LRU of CL);
17:     else
18:         victim ← E(LRU of DL);
19:     end if
20: end if
21: return victim;
```

by the eviction process. Compared with the LRU algorithm, the GALRU requires additional operations for checking the EPR and selecting the victim entry. Because the size of the metadata for the EPR and the number of candidates for the victim entry (LRU buffer entry of the clean or dirty list) are constant, the additional time complexity is O(1).

V. CASE STUDY

This section presents case studies based on the ideas of the GALRU to improve prevalent buffer management schemes. The eviction policy applied in units of the SR group can be used to overcome the limitation of current buffer management schemes. The subsections below present case studies that apply the victim selection algorithm of the GALRU to the DPW-LRU [28] and PT-LRU [22]. The impact of each case on the performance of the flash storage system is described in Section 6.

A. DPWGA-LRU

The DPWGA-LRU is a buffer management scheme that applies the victim selection algorithm of the GALRU to the eviction process of the DPW-LRU. The DPW-LRU divides the buffer into two regions: a working region and an exchange region. The working region plays a role similar to that of the common region in the GALRU, and the exchange region acts similarly to the victim region in the GALRU. The page weight, which is used to select the target entry for region migration, reflects the characteristics of the data, such as the interval of temporal locality, cost of eviction, and recency. Therefore, the DPW-LRU can improve the buffer hit ratio and reduce the number of flash write operations. However, the DPW-LRU has a limitation whereby non-critical sub-requests lead to a waste of clean entries because it selects the victim entry by using the LRU algorithm. To improve the eviction process of the DPW-LRU, the DPWGA-LRU divides the exchange region into a clean list and a dirty list, and replaces the LRU algorithm with the victim selection algorithm of the GALRU. Algorithm 3 is the eviction algorithm of the DPWGA-LRU. Except for the second and third lines, Algorithm 3 is identical to the eviction algorithm of the DPW-LRU.

Algorithm 3 Eviction Process of DPWGA-LRU

WR: working region; VR: exchange region; R: selected region; P_{weight}: weight of a page; Map_{weight}: weight map of pages;

Input: R

Output: reference of *victim*;

```
1:  if R ∈ ER then
2:      victim ← Victim_Selection_of_GALRU();
3:      return victim
4:  else
5:      Run the DPW-LRU strategy in WR
6:      for pages ∈ [0, w] of WR do
7:          P_weight ← DPW(page);
8:          Map_weight.put(page, P_weight);
9:      end for
10:     for page ∈ Map_weight do
11:         if page < W_min then
12:             victim ← page;
13:             W_min ← Map_weight.get(page);
14:         end if
15:     end for
16:     return victim;
17: end if
```

B. PTGA-LRU

The PTGA-LRU is a buffer management scheme that applies the victim selection algorithm of the GALRU to the eviction process of the PT-LRU. The PT-LRU manages the buffer by dividing it into three lists: a cold clean linked list (LC), a cold dirty linked list (LD), and a mixed LRU linked list (LH). The LC, LD, and LH lists play roles similar to those of the clean, dirty, and common lists in the GALRU, respectively. The PT-LRU can reduce the number of flash write operations because it can create a situation in which the victim entries are selected from the LH list, even if the LD list is not empty, by using a probability-based algorithm. However, the

Algorithm 4 Eviction Process of PTGA-LRU

LC; cold clean queue; LD: cold dirty queue; LH: mixed region; E(X): buffer entry of X; RID: ID of I/O request; *victim*: victim entry

Input: RID

Output: Reference *victim*;

1: **if** PID of EPR == reset **then**
2:     **if** Size of the SR group of RID $\leq$ size of LC **then**
3:         Set PID of EPR to CO;
4:     **else if** Size of the SR group of RID $\leq$ size of LD **then**
5:         Set PID of EPR to DO;
6:     **else**
7:         Set PID of EPR to MA;
8:     **end if**
9: **end if**
10: **if** PID of EPR == CO **then**
11:     *victim* $\leftarrow$ E(LRU of LC);
12: **else if** PID of EPR == DO **then**
13:     *victim* $\leftarrow$ E(LRU of LD);
14: **else**
15:     **if** LC is not NULL **then**
16:         *victim* $\leftarrow$ E(LRU of LC);
17:     **else**
18:         Calculate *Replace-Flag* by *pro*; //(0.5$<$pro$<$1)
19:         **if** *Replace-Flag* == 1 and LD is not NULL **then**
20:             *victim* $\leftarrow$ E(LRU of LD);
21:         **else**
22:             *victim* $\leftarrow$ PT-LRU_EvictHot();
23:         **end if**
24:     **end if**
25: **end if**
26: **return** *victim*;

PT-LRU cannot prevent the problem whereby non-critical sub-requests waste clean entries because it prefers to use the LC list over other lists for selecting the victim entry when it is not empty. To apply the victim selection algorithm of the GALRU to the eviction process of the PT-LRU, the PTGA-LRU modifies the eviction policies of the GALRU. The CO policy selects the LRU buffer entry of the LC list as victim entry, the DO policy selects the LRU buffer entry of the LD list as the victim entry, and the MA policy selects the victim entry using the eviction algorithm of the PT-LRU. Algorithm 4 shows the eviction process of the PTGA-LRU. Notably, the PTGA-LRU replaces the eviction algorithm of the PT-LRU, that is, lines 8–15 of the page management algorithm of the PT-LRU, with Algorithm 4. Furthermore, lines 15–24 of Algorithm 4 are the same as the eviction algorithm of the PT-LRU.

VI. PERFORMANCE EVALUATION

In this section, the impact of the GALRU on the performance of a flash storage system is analyzed using a trace-driven simulator. The GALRU is compared with the LRU, CFLRU, PT-LRU, and DPW-LRU by conducting several experiments. The PTGA-LRU and the DPWGA-LRU, mentioned in the above case studies, were compared with the PT-LRU and the DPW-LRU, respectively. To analyze the impact of buffer management schemes on the performance of a flash storage system that uses internal parallelism, the allocation pattern of buffer entries was measured in addition to the buffer hit ratio and the number of flash write operations.

A. EXPERIMENTAL SETUP

An in-house simulator that supports the simulation of multi-channel, multi-way architectures was used for the experiments. Even though Flash-DBsim [44] has been used for experiments on many buffer management schemes, it is not suitable for experiments with the GALRU because it does not support internal parallelism. To investigate the impact of the buffer management schemes on I/O latency when sub-requests are processed in parallel, we implemented a trace-driven in-house simulator based on Flash-DBsim to simulate various multi-channel, multi-way architectures. Real I/O traces generated by different types of applications were used for the experiments. Table 1 shows information on the I/O traces. Financial1 and Websearch1 were collected by the Storage Performance Council (SPC) [45]. The other trace files were collected through research [46] related to the I/O characteristics of smartphones.

TABLE 1. I/O trace specifications.

Trace	Write(%)	Read(%)	Description
Financial1	76.84	23.16	Running OLTP application
Websearch1	0.02	99.98	Running a search engine
Messaging	97.30	2.70	Receiving/sending/viewing messages
Music	52.80	47.20	Listening to songs on the smartphone
Booting	33.07	66.93	Smartphone booting process
Amazon	99.75	11.25	Mobile online shopping
RadioWebBrowse	72.01	27.99	Listening to online radio

TABLE 2. Simulation settings.

Attribute	Configuration
Page size	2 KB
Block size	128 KB
Read latency	25 μs
Write latency	200 μs
Erase latency	1500 μs
Channel data bus speed	800 MT/s
Channel data bus bit width	8
Way arbitration policy	Round robin

Table 2 shows the settings used by the in-house simulator. Settings related to latency and size were the same as those used in the PT-LRU [22] and the DPW-LRU [28]. The other settings were added to simulate the multi-channel, multi-way architecture. The transfer speed of the channel data bus was 800 MT/s, which is the maximum speed of the data interface supported by ONFI 4.0 [12]. The round-robin method was used as an arbitration policy in case of competition between ways of using the channel data bus. The simulator did not support multi-plane operation.

The main settings used by each buffer management scheme are as follows: The LRU was implemented to manage all data belonging to the buffer using the LRU algorithm. The CFLRU used half of the buffer as a clean-first window, and the PT-LRU and PTGA-LRU set the default value of *pro* to 0.8, which was the most suitable for reducing the number of write operations in experiments on the PT-LRU [22]. Here, *pro* is the factor that determines the probability of evicting cold and dirty data. The DPW-LRU and DPWGA-LRU used half of the buffer as the working region and the other half as the exchange region. The GALRU used half of the buffer as the common region and the other half as the victim region.

B. I/O LATENCY

1) IMPACT OF INTERNAL PARALLELISM ON I/O LATENCY

This section analyzes the impact of the GALRU on I/O latency depending on whether internal parallelism is used. The GALRU was designed based on the assumption that a sub-request using a dirty entry is likely to become a critical sub-request. This assumption was established when sub-requests were processed in parallel. When a flash storage system did not utilize internal parallelism, the sub-request that was processed last became a critical sub-request because they were processed sequentially.

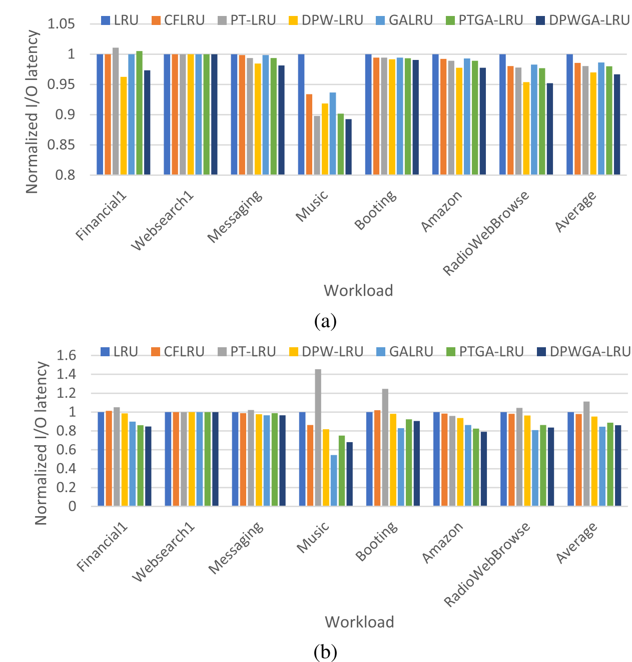


FIGURE 5. Normalized I/O latency: (a) one-channel, one-way configuration; (b) eight-channel, eight-way configuration.

Figure 5 shows the normalized I/O latency. The I/O latency of each scheme was normalized by the I/O latency of the LRU, and the size of the buffer was 4 MB. Because the simulator did not support plane-level parallelism, the one-channel, one-way configuration did not utilize internal parallelism. In this configuration, because improving the buffer hit ratio or reducing the number of flash write operations was important for reducing I/O latency, the GALRU was not effective in reducing the I/O latency compared with the other schemes considered. The results of measurement of the buffer hit ratio and number of flash write operations are described in Sections 6-D and 6-E, respectively. In the eight-channel, eight-way configuration that could adequately use internal parallelism, the I/O latency of the GALRU was lower than that of the other schemes in most cases. Compared with those of the LRU, CFLRU, PT-LRU, and DPW-LRU, the I/O latencies of the GALRU were 15.5%, 13.7%, 24.0%, and 11.2% lower, respectively. Because the GALRU improved the likelihood that clean entries, which were wasted by the other schemes, would be used to reduce I/O latency, it yielded better performance than the other schemes. To analyze changes in the allocation pattern of clean entries, we classified I/O requests depending on the dirtiness of the allocated buffer entries. The results are described in Section 6-C.

Moreover, the basic idea of GALRU was effective in improving the schemes considered above. In the eight-channel, eight-way configuration, the I/O latencies of GALRU-based schemes introduced in Section 5 was lower than those of the PT-LRU and the DPW-LRU. The I/O latency of the PTGA-LRU was 20.1% lower than that of the PT-LRU, and the I/O latency of the DPWGA-LRU was 9.5% lower than that of the DPW-LRU. The application of GALRU does not significantly affect the eviction policy of these schemes because it can be applied to them without major modifications, as explained in Section 5. Therefore, clean entries were managed efficiently without significant changes to the buffer hit ratio or the number of flash write operations. As a result, the PTGA-LRU and DPWGA-LRU were able to further reduce I/O latency over the PT-LRU and DPW-LRU, respectively.

2) IMPACT OF BUFFER SIZE ON I/O LATENCY

Figure 6 shows the normalized I/O latency of the eight-channel, eight-way configuration using buffers of various sizes. Regardless of the size of the buffer, the I/O latencies of the GALRU and GALRU-based schemes were lower than those of the other schemes. The GALRU reduced I/O latency compared with the average I/O latency of the four schemes considered: 16.0% for a 1 MB buffer, 16.1% for a 2 MB buffer, 16.1% for a 4 MB buffer, and 15.0% for an 8 MB buffer. The PTGA-LRU reduced I/O latency compared with the average I/O latency of the PT-LRU as well: 18.2% for a 1 MB buffer, 20.0% for a 2 MB buffer, 18.1% for a 4 MB buffer, and 17.8% for an 8 MB buffer. The DPWGA-LRU reduced the I/O latency compared with the average I/O latency of DPW-LRU by 14.0% for a 1 MB buffer, 12.0% for a 2 MB buffer, 9.7% for a 4 MB buffer, and 9.8% for an 8 MB buffer.

3) IMPACT OF WORKLOAD INTENSITY ON I/O LATENCY

This section analyzes the impact of the GALRU on I/O latency according to the intensity of the workload. When the intensity of the workload was high, many sub-requests

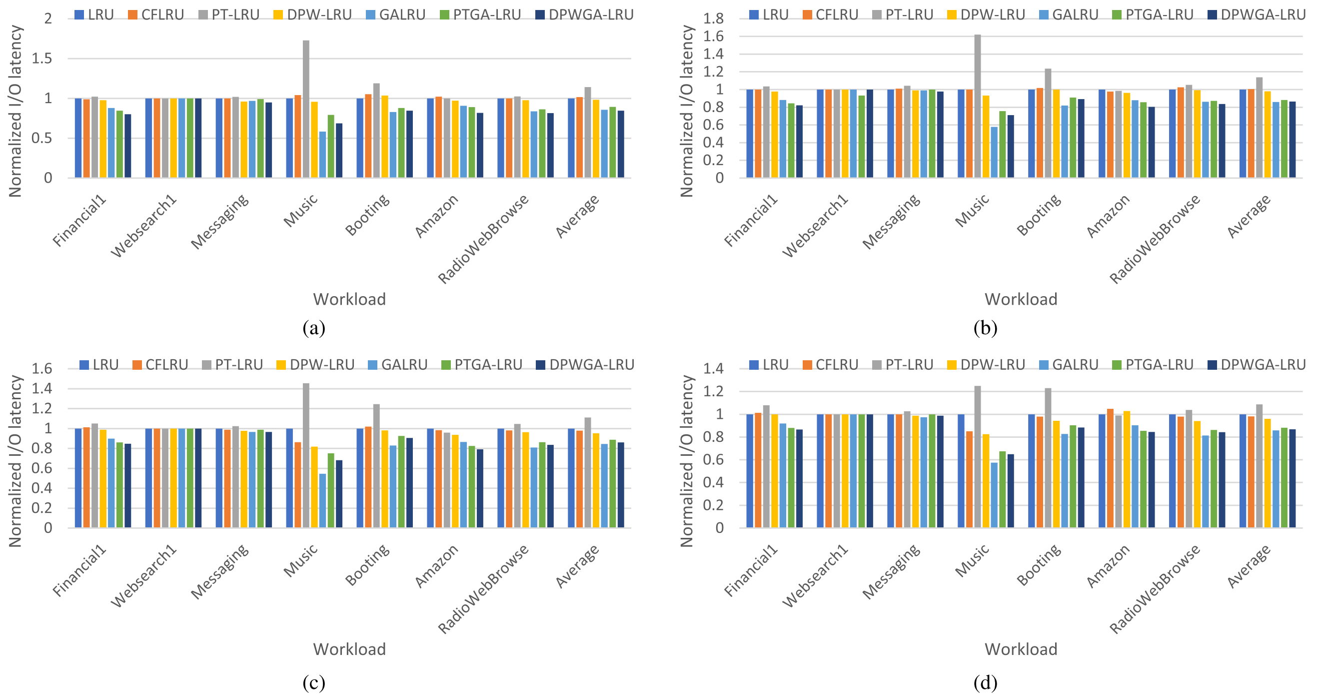


FIGURE 6. Normalized I/O latency of eight-channel, eight-way configuration: (a) 1 MB buffer; (b) 2 MB buffer; (c) 4 MB buffer; (d) 8 MB buffer.

were accumulated in request queues inside the flash storage system owing to the slow operation of the flash device. In this case, because the impact of the preceding sub-request on the processing time of the subsequent sub-request increased, it was difficult to predict the sub-request that was critical. The impact of the GALRU on I/O latency decreased as the intensity of the workload increased because the possibility that a sub-request using a dirty entry became a critical sub-request decreased. To analyze the impact of the GALRU in such cases, we measured I/O latency by ignoring the timestamp of the I/O trace files and adjusting the number of outstanding I/O requests.

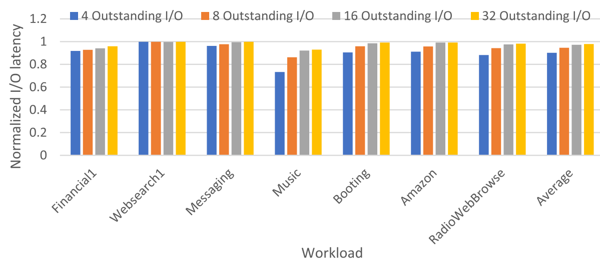


FIGURE 7. Normalized I/O latency of eight-channel, eight-way configuration.

Figure 7 shows the normalized I/O latency depending on the number of outstanding I/O requests. The I/O latency of the GALRU was normalized by that of the LRU, and the size of the buffer was 4 MB. As the number of outstanding I/O requests increased, the impact of the GALRU in terms of reducing I/O latency decreased. The I/O latencies of the GALRU were 9.8%, 5.3%, 2.7%, and 2.1% lower than those of the LRU when the numbers of outstanding I/O requests were 4, 8, 16, and 32, respectively. Considering that the intensity of the workload on systems [47], [48] is kept low in practice for a considerable amount of time, the GALRU is sufficiently effective in improving performance.

C. ALLOCATION PATTERN OF BUFFER ENTRIES

To analyze changes in the allocation pattern of buffer entries, we classified I/O requests depending on the type of allocated buffer entries. Table 3 shows features of each type of I/O request. The number of each was measured by processing all I/O requests generated by the workloads and classifying the results.

TABLE 3. Types of I/O request.

Type		Description
Full-hit (FH)		Sub-requests of the FH-type I/O request did not trigger the eviction process.
Miss-clean (MC)		Sub-requests of the MC-type I/O request did not use a dirty entry as the victim entry for the eviction process.
Miss-dirty (MD)	MDC	Sub-requests of the MDC-type I/O request used a dirty entry and a clean entry as the victim entry
	MDD	Sub-requests of the MDD-type I/O request used only a dirty entry as the victim entry

Figure 8 shows the ratio of each type of I/O request to total I/O requests when the flash storage system used a 4 MB buffer. The GALRU reduced the number of MD-type I/O requests by 5.64% on average compared with the four

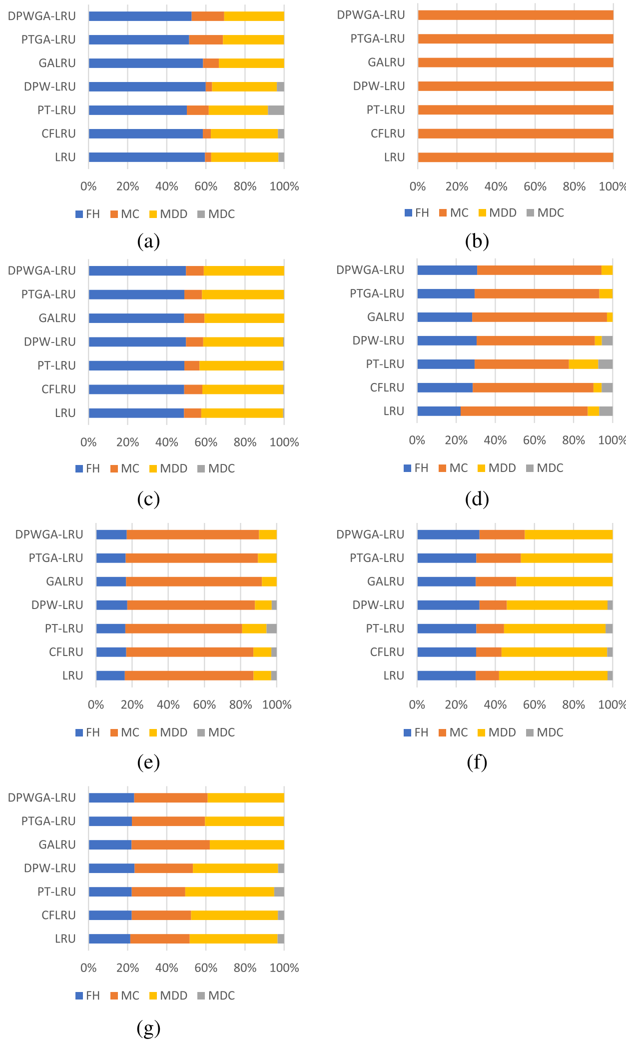


FIGURE 8. I/O request type for various workloads: (a) Financial1; (b) Websearch1; (c) Messaging; (d) Music; (e) Booting; (f) Amazon; (g) RadioWebBrowse.

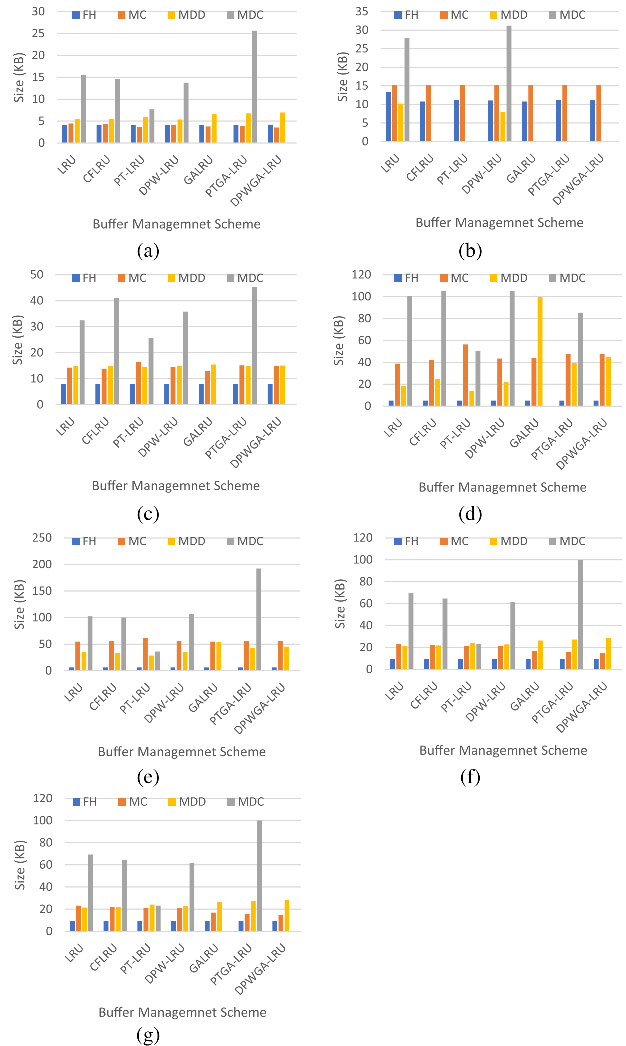


FIGURE 9. I/O request size for various workloads: (a) Financial1; (b) Websearch1; (c) Messaging; (d) Music; (e) Booting; (f) Amazon; (g) RadioWebBrowse.

schemes considered. The PTGA-LRU reduced the number of MD-type I/O requests by 7.36% on average compared with the PT-LRU, and the DPWGA-LRU reduced the number of MD-type I/O requests by 4.12% on average compared with the DPW-LRU. Because the eviction process for writing dirty data to the flash device delayed the I/O operations for the requested data, the I/O latency of the MD-type I/O request was likely to be higher than those of the FH-type and the MC-type I/O requests. If the number of the MD-type I/O requests decreased, that of the FH-type or the MC-type I/O requests increased. As a result, as the number of the MD-type I/O requests decreases, the average I/O latency decreases because the FH-type or MC-type I/O requests have a lower latency than the MD-type I/O request.

Because of the difference in size between the types of I/O requests, the GALRU was able to reduce the number of MD-type I/O requests to a greater extent than expected. Even though it is designed to reduce the number of the MDC-type I/O requests, the number of MDD-type I/O requests was also reduced as shown in Figure 8. This result is related to the difference in size between I/O request types. The size of the I/O request refers to the size of data requested by the given I/O request. Figure 9 shows the size of each type of I/O request. The size of the MDC-type I/O request was larger than that of the other types considered. This means that more MC-type I/O requests could be generated than the reduced number of MDC-type I/O requests. Most of the increased MC-type I/O requests underwent conversion from MDD-type I/O requests. Except for the read-dominant *Websearch1* workload, the size of the MDD-type I/O requests of the GALRU was greater than that of the four schemes considered. Furthermore, the size of the MC-type I/O requests of the GALRU was smaller than those of these schemes. This result means that the GALRU tended to convert large MDC-type I/O requests into MDD-type I/O requests and small MDD-type I/O requests into MC-type I/O requests.

Figure 10 shows the ratio of each type of I/O request to total I/O requests for buffers of various sizes.

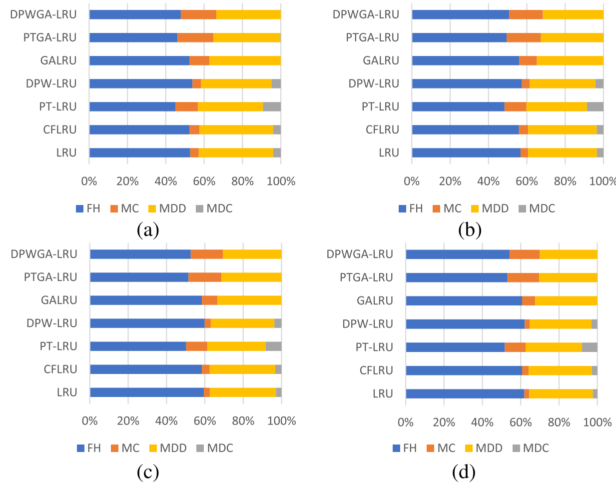


FIGURE 10. I/O request type for Financial1 workload: (a) 1 MB buffer; (b) 2 MB buffer; (c) 4 MB buffer; (d) 8 MB buffer.

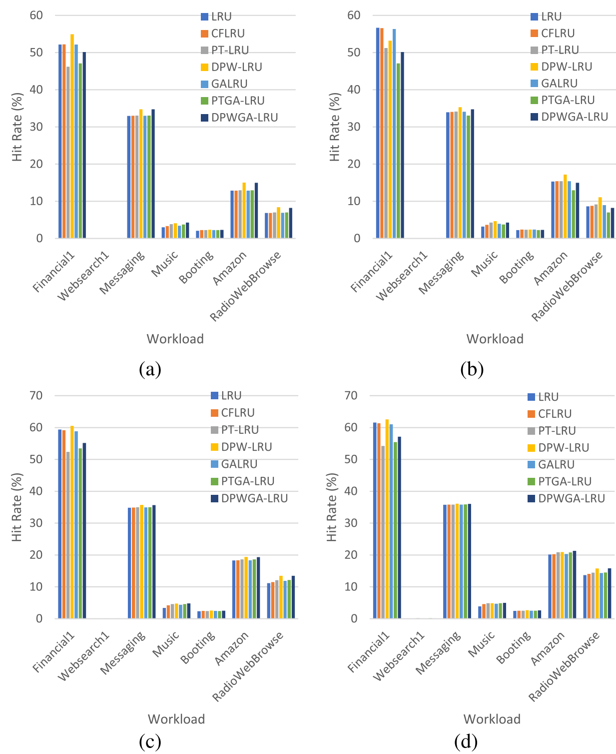


FIGURE 11. Buffer hit ratio: (a) 1 MB buffer; (b) 2 MB buffer; (c) 4 MB buffer; (d) 8 MB buffer.

Regardless of buffer size, the GALRU and GALRU-based schemes reduced the number of MD-type I/O requests. The GALRU reduced them by 5.23% for a 1 MB buffer, 4.67% for a 2 MB buffer, 4.20% for a 4 MB buffer, and 3.59% for an 8 MB buffer, compared with the other schemes on average. As the size of the buffer increased, the number of FH-type I/O requests increased because the buffer hit ratio increased. Conversely, as the size of the buffer increased, the number of MD-type I/O requests that could be reduced by the GALRU decreased. The PTGA-LRU reduced the number of MD-type I/O requests of the PT-LRU: 8.02% for the 1 MB buffer, 7.59% for the 2 MB buffer, 7.30% for the 4 MB buffer, and 7.03% for the 8 MB buffer. The DPWGA-LRU reduced the number of MD-type I/O requests of the DPW-LRU: 7.88% for the 1 MB buffer, 6.97% for the 2 MB buffer, 6.22% for the 4 MB buffer, and 5.37% for the 8 MB buffer.

D. BUFFER HIT RATIO

Figure 11 shows the buffer hit ratio for buffers of various sizes. As the size of the buffer increased, the buffer hit ratio increased because data with a high temporal locality remained in the buffer for a long time. The buffer hit ratio of the GALRU was lower than that of other schemes in several cases because the GALRU could evict hot data to leave clean data in the buffer. However, it did not significantly reduce the buffer hit ratio because the situation wherein dirty data were evicted instead of clean data did not occur often. For example, in the 4 MB buffer, the buffer hit ratio of the GALRU was 0.79% lower on average than that of the DPW-LRU, which had the highest buffer hit ratio. However, as confirmed in Sections 6-B and 6-C, the GALRU was more effective in reducing the I/O latency of a flash storage system that utilized internal parallelism than other schemes with higher buffer hit ratios because the allocation pattern of buffer entries had a significant effect on I/O latency.

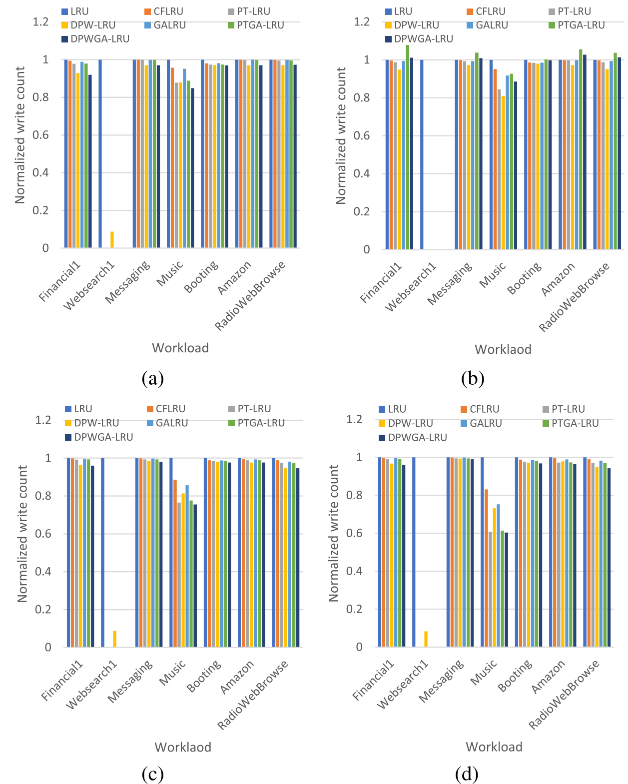


FIGURE 12. Number of flash write operations: (a) 1 MB buffer; (b) 2 MB buffer; (c) 4 MB buffer; (d) 8 MB buffer.

E. NUMBER OF FLASH WRITE OPERATIONS

Figure 12 shows the normalized number of flash write operations based on the number of flash write operations of the LRU for buffers of various sizes. As the size of the buffer

increased, the number of flash write operations decreased because dirty data remained in the buffer for a longer time. Even though all schemes, except for the LRU, employed the clean-first method, the scheme with the lowest number of flash write operations differed depending on the workload because of differences in the eviction algorithms used. Because the GALRU did not use the clean-first method when the amount of clean data in the buffer was insufficient, its number of flash write operations was higher than those of other schemes in several cases. However, as confirmed in Sections 6-B and 6-C, the GALRU was more effective in reducing the I/O latency of a flash storage system that utilizes internal parallelism than other schemes with a smaller number of flash write operations because the allocation pattern of buffer entries had a large effect on I/O latency.

VII. CONCLUSION

To improve the I/O performance of flash storage systems that utilize internal parallelism, we proposed a buffer management scheme called the GALRU. It applies an eviction policy in units of sub-request groups. The GALRU avoids allocating buffer space that contains both clean data and dirty data to an I/O request to efficiently use the space that contains clean data, where this helps reduce I/O latency. The GALRU can unify the dirtiness of data contained in the buffer space such that it is allocated to an I/O request, because it groups sub-requests derived from the same I/O request and applies the same eviction policy to sub-requests belonging to the same group. To analyze the impact of the GALRU on I/O latency, we classified I/O requests into several types according to the results of the eviction process. The results of experiments showed that the GALRU increases the number of I/O requests with low latency by reducing the number of requests that evict both clean and dirty data from the buffer.

REFERENCES

[1] NVM Express. (2019). *Nvm Express Specification 1.4*. [Online]. Available: http://nvmexpress.org/resources/specifications/

[2] SATA-IO. (2018). *Serial Ata Revision 3.4 Specification*. [Online]. Available: https://sata-io.org/developers/purchase-specification

[3] JEDEC. (2020). *Universal Flash Storage (Ufs), Version 3.1*. [Online]. Available: https://www.jedec.org/standards-documents/focus/flash/universal-flash-s%torage-ufs

[4] S.-y. Park, E. Seo, J.-Y. Shin, S. Maeng, and J. Lee, "Exploiting internal parallelism of flash-based SSDs," *IEEE Comput. Archit. Lett.*, vol. 9, no. 1, pp. 9–12, Jan. 2010.

[5] F. Chen, R. Lee, and X. Zhang, "Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 266–277.

[6] F. Chen, B. Hou, and R. Lee, "Internal parallelism of flash memory-based solid-state drives," *ACM Trans. Storage*, vol. 12, no. 3, pp. 1–39, Jun. 2016.

[7] N. Elyasi, M. Arjomand, A. Sivasubramaniam, M. T. Kandemir, C. R. Das, and M. Jung, "Exploiting intra-request slack to improve SSD performance," in *Proc. 32nd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, Apr. 2017, pp. 375–388.

[8] J. Cui, Y. Zhang, W. Wu, J. Yang, Y. Wang, and J. Huang, "DLV: Exploiting device level latency variations for performance improvement on flash memory storage systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 37, no. 8, pp. 1546–1559, Aug. 2018.

[9] W. Zhang, Q. Cao, H. Jiang, and J. Yao, "Improving overall performance of TLC SSD by exploiting dissimilarity of flash pages," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 2, pp. 332–346, Feb. 2020.

[10] S. Nie, Y. Zhang, W. Wu, C. Zhang, and J. Yang, "DIR: Dynamic request interleaving for improving the read performance of aged SSDs," in *Proc. IEEE Non-Volatile Memory Syst. Appl. Symp. (NVMSA)*, Aug. 2019, pp. 1–6.

[11] W. Zhang, Q. Cao, H. Jiang, and J. Yao, "PA-SSD: A page-type aware TLC SSD for improved Write/Read performance and storage efficiency," in *Proc. Int. Conf. Supercomputing*, Jun. 2018, pp. 22–32.

[12] Open NAND Flash Interface Working Group. (2014). *Onfi 4.0 Specification*. [Online]. Available: http://www.onfi.org/specifications

[13] M. Abraham, "Nand flash architecture and specification trends," *Flash Memory Summit*, vol. 4, pp. 1–18, Aug. 2012.

[14] Y. Li and K. N. Quader, "NAND flash memory: Challenges and opportunities," *Computer*, vol. 46, no. 8, pp. 23–29, Aug. 2013.

[15] 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. CASES*, 2006, pp. 234–241.

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

[17] 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.

[18] 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.

[19] 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.

[20] M. Lin, S. Chen, and Z. Zhou, "An efficient page replacement algorithm for NAND flash memory," *IEEE Trans. Consum. Electron.*, vol. 59, no. 4, pp. 779–785, Nov. 2013.

[21] M. Lin, S. Chen, G. Wang, and T. Wu, "HDC: An adaptive buffer replacement algorithm for NAND flash memory-based databases," *Optik*, vol. 125, no. 3, pp. 1167–1173, Feb. 2014.

[22] 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.

[23] C. Li, D. Feng, Y. Hua, W. Xia, and F. Wang, "Gasa: A new page replacement algorithm for NAND flash memory," in *Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS)*, Aug. 2016, pp. 1–9.

[24] M. Lin, Z. Yao, and T. Huang, "F-LRU: An efficient buffer replacement algorithm for NAND flash-based databases," *Optik*, vol. 127, no. 2, pp. 663–667, Jan. 2016.

[25] J. He, G. Jia, G. Han, H. Wang, and X. Yang, "Locality-aware replacement algorithm in flash memory to optimize cloud computing for smart factory of industry 4.0," *IEEE Access*, vol. 5, pp. 16252–16262, 2017.

[26] 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.

[27] X. Wu, D. Cai, and S. Guan, "A multiple LRU list buffer management algorithm," *IOP Conf. Ser., Mater. Sci. Eng.*, vol. 569, Aug. 2019, Art. no. 052002.

[28] 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.

[29] 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.

[30] E. H. Nam, B. S. J. Kim, H. Eom, and S. L. Min, "Ozone (O3): An out-of-order flash memory controller architecture," *IEEE Trans. Comput.*, vol. 60, no. 5, pp. 653–666, May 2011.

[31] 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.

[32] D. Wei, Y. Gong, L. Qiao, and L. Deng, "A hardware-software co-design experiments platform for NAND flash based on zynq," in *Proc. IEEE 20th Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, Aug. 2014, pp. 1–7.

[33] J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, "Cosmos+ OpenSSD: Rapid prototype for flash storage systems," *ACM Trans. Storage*, vol. 16, no. 3, pp. 1–35, Aug. 2020.

[34] S.-H. Park, S.-H. Ha, K. Bang, and E.-Y. Chung, "Design and analysis of flash translation layers for multi-channel NAND flash-based storage devices," *IEEE Trans. Consum. Electron.*, vol. 55, no. 3, pp. 1392–1400, Aug. 2009.

[35] H. Jung, S. Jung, and Y. Song, "Architecture exploration of flash memory storage controller through a cycle accurate profiling," *IEEE Trans. Consum. Electron.*, vol. 57, no. 4, pp. 1756–1764, Nov. 2011.

[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] S.-H. Park, J.-W. Park, S.-D. Kim, and C. C. Weems, "A pattern adaptive NAND flash memory storage structure," *IEEE Trans. Comput.*, vol. 61, no. 1, pp. 134–138, Jan. 2012.

[38] S. K. Park, Y. Park, G. Shim, and K. H. Park, "CAVE: Channel-aware buffer management scheme for solid state disk," in *Proc. ACM Symp. Appl. Comput. SAC*, 2011, pp. 346–353.

[39] H. Sun, G. Chen, J. Huang, X. Qin, and W. Shi, "CalmWPC: A buffer management to calm down write performance cliff for NAND flash-based storage systems," *Future Gener. Comput. Syst.*, vol. 90, pp. 461–476, Jan. 2019.

[40] S. T. On, S. Gao, B. He, M. Wu, Q. Luo, and J. Xu, "FD-buffer: A cost-based adaptive buffer replacement algorithm for FlashMemory devices," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2288–2301, Sep. 2014.

[41] Q. Wei, C. Chen, and J. Yang, "CBM: A cooperative buffer management for SSD," in *Proc. 30th Symp. Mass Storage Syst. Technol. (MSST)*, Jun. 2014, pp. 1–12.

[42] Y. Yao, X. Kong, J. Zhou, X. Xu, W. Feng, and Z. Liu, "An advanced adaptive least recently used buffer management algorithm for SSD," *IEEE Access*, vol. 7, pp. 33494–33505, 2019.

[43] D. Kim, K. H. Park, and C.-H. Youn, "SUPA: A single unified read-write buffer and Pattern-Change-Aware FTL for the high performance of multi-channel SSD," *ACM Trans. Storage*, vol. 13, no. 4, pp. 1–30, Dec. 2017.

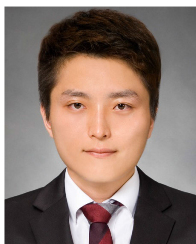
[44] X. Su, P. Jin, X. Xiang, K. Cui, and L. Yue, "Flash-DBSim: A simulation tool for evaluating flash-based database algorithms," in *Proc. 2nd IEEE Int. Conf. Comput. Sci. Inf. Technol.*, 2009, pp. 185–189.

[45] Storage Performance Council. (2002). *SPC Trace File Format Specification*. [Online]. Available: http://traces.cs.umass.edu/index.php/Storage/Storage

[46] D. Zhou, W. Pan, W. Wang, and T. Xie, "I/O characteristics of smartphone applications and their implications for eMMC design," in *Proc. IEEE Int. Symp. Workload Characterization*, Oct. 2015, pp. 12–21.

[47] S. Sankar and K. Vaid, "Storage characterization for unstructured data in online services applications," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Oct. 2009, pp. 148–157.

[48] K. Han and D. Shin, "Command queue-aware host I/O stack for mobile flash storage," *J. Syst. Archit.*, vol. 109, Oct. 2020, Art. no. 101758.

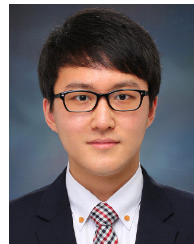


JAEWOOK KWAK received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2015. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University. His research interests include computer architecture, embedded systems, and NAND flash-based storage systems.

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.

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.

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.



GYEONGYONG 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 NAND flash memories.

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 at 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 the Technical Committee (DiSPS) in the 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 for 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.