Hindawi
Discrete Dynamics in Nature and Society
Volume 2022, Article ID 3376904, 11 pages
https://doi.org/10.1155/2022/3376904

Research Article

URM: A Unified RAM Management Scheme for NAND Flash Storage Devices

A. Xiaochang Li[1], **B. Jichen Chen**[2], **C. Zhengjun Zhai**[1], **D. Mingchen Feng**[3] and **E. Xin Ye**[4]

[1]*School of Computer Science and Engineering, Northwestern Polytechnical University, Xi'an 710072, China*
[2]*Xi'an Microelectronics Technology Institute, Xi'an, China*
[3]*College of Information Engineering, Northwest A&F University, Xi'an, China*
[4]*Xi'an Technological University, Xi'an, China*

Correspondence should be addressed to C. Zhengjun Zhai; zhaizjun@nwpu.edu.cn and D. Mingchen Feng; mingcheng89@126.com

Received 30 December 2021; Accepted 4 May 2022; Published 31 May 2022

Academic Editor: Wei Wang

In NAND flash storage devices, the random access memory (RAM) is composed of a data buffer and mapping cache that play critical roles in storage performance. Furthermore, as the capacity growth rate of RAM chips lags far behind that of flash memory chips, determining how to take advantage of precious RAM is still a crucial issue. However, most existing buffer management studies on storage devices report performance degradation since these devices cannot refine reference regularities such as sequential, hot, or looping data patterns. In addition, most of these studies focus only on separately managing the data buffer or mapping cache. Compared with the existing buffer/cache management schemes (BMSs), we propose a unified RAM management (URM) scheme for not only the mapping cache but also the data buffer in NAND flash storage devices. URM compresses the mapping table to save memory space, and the remaining dynamic RAM space is used for the data buffer. For the data buffer part, we utilize the program counter-technique in the host layer that provides automatic pattern recognition for different applications, in contrast to existing BMSs. The program counter-technique in our design is able to distinguish four patterns. According to these patterns, the data buffer is divided into four size-adjustable zones. Therefore, our approach is linked to multimodal data and used in a data-intensive system. In particular, in URM, we use a multivariate classification to predict prefetching length in mapping buffer management. Our multivariate classification is transformed into multiple binary classifications (logistic regressions). Finally, we extensively evaluate URM using various realistic workloads, and the experimental results show that, compared with three data buffer management schemes, CFLRU, BPLRU, and VBBMS, URM can improve the hit ratio of data buffer and save response time by an average to 32% and 18%, respectively.

1. Introduction

Storage devices based on NAND flash memory have been accepted due to their obvious advantages over hard disk drives (HDDs), i.e., lower power consumption, lower access latency, and smaller size. Currently, with the character of increasing bit density and decreasing bit cost of NAND flash chip, it is growing rapidly in the storage market. With aggressive scaling of the chip density, flash memory chips have developed rapidly from the single-level cell (SLC) to multilevel cell (MLC) and V-NAND flash memory [1]. However, as we mentioned, the capacity growth rate of RAM chips lags far behind that of flash memory chips. In particular, for embedded systems and portable devices [2], there is not enough space for more RAM chips. Regardless of how large the capacity of a RAM chip can be, it is still unable to meet the increasing storage size requirement. Therefore, studying the RAM management scheme is critical in NAND flash chip-based storage devices.

In addition to the data buffer, the flash translation layer (FTL) in the mapping cache is an indispensable part of the NAND flash storage device, which keeps mapping information (i.e., an entire mapping table) transforming virtual addresses created by upper layers (i.e., file systems in the

host) to physical addresses on the flash chips [3, 4]. In conclusion, RAM in the flash storage device is composed of both data buffer and mapping cache, which plays several roles, including (i) accommodating data in buffer to benefit storage performance, such as random write performance, (ii) storing the mapping table in the mapping cache to emulate the functionality of a normal block device, and (iii) improving the flash memory lifetime due to the flash memory programming or erasing limitation.

How to manage RAM well is a key concept for the flash storage device. In particular, it is suitable for the complex and data-intensive system. However, most of the existing studies focus on either data buffer or mapping cache management. For instance, there exist several strategies only focusing on data buffer management. According to the granularity type, they are classified into a page unit, a block unit, or both page and block unit buffer management schemes [5–10]. Most of them are based on the widely used least recently used (LRU) to distinguish the accessed data patterns, either spatial or temporal localities. Note that, although LRU is classical and adapts very well to the changes in the workload, the main drawbacks of the LRU scheme are that it cannot refine access pattern regularities and its recognition ability is limited by the buffer size, which yields degraded performance [11].

In this design, we devise a novel system-level buffer management policy and unify buffer and mapping cache managements, which is called URM. It adopts the program counter-technique to automatically exploit workload patterns with fine-grained granularity. In particular, for a key prediction (prefetching length prediction) of buffer management, a multivariate classification is used. In URM, our multivariate classification is transformed into multiple binary classifications (logistic regressions) [12]. For a detailed description, refer to Section 4.2.

The success in using the program counter in branch prediction was developed, and the PC information has been widely used in other predictor designs in computer architecture. Numerous PC-based predictors have been designed to optimize energy [13], cache management [14, 15], memory prefetching [14], operating system design [16], and NAND flash storage devices [17] areas.

A brief workflow is described as follows. The pattern identification process of URM occurs at the host layer. Then, the predicted data patterns will be transferred to the FTL of a flash storage device without any further pattern identification cost in RAM of the flash memory device. With the pattern prediction, URM divides buffer space into four size-adjustable zones according to its new four pattern classifications. Furthermore, we compress the mapping table to reduce the mapping cache size. The dynamic RAM management can save more RAM space. Compared with existing buffer management policies, our design makes the contributions as follows:

(i) To our knowledge, this is the first time to propose a unified memory management policy that addresses the data buffer together with the compressed mapping cache.

(ii) Compared with the traditional spatial and temporal locality approaches, our new pattern identification technique refines pattern classifications of sequential, looping, clustered hot, and random patterns, which supports different pattern-based management schemes.

(iii) The data buffer has been divided into four size-adjustable zones, which provide an efficient and flexible buffer management scheme.

As mentioned before, the DRAM of SSD will be used for two purposes, data buffer and mapping cache management. However, traditional DRAM management research divides DRAM into two parts to accommodate data buffer and mapping cache, but there are no efficient unified DRAM management algorithms. Furthermore, the increasing capacity of NAND flash chip requires more mapping cache space, which urges researchers to compact mapping cache and balance the size of data buffer and mapping cache in DRAM. Therefore, we put forward a unified DRAM management algorithm. How to design and deploy our PC-based algorithm has been described in this article.

The rest of this study is organized as follows. We explain the existing techniques in Section 2. Section 3 describes the key ideas and motivations behind URM. The design of URM is shown in Section 4. Section 5 provides experimental results to prove the design effectiveness. Finally, a summary has been concluded in Section 6.

2. Background

The data buffer is a standard component for the commercial NAND flash storage device, which not only helps to delay the submission of random writes but can also rearrange random writes into sequential writes. However, the performance of the data buffer is affected by the data access mode. Let us take the random pattern as an example for analysis. Random writing is much slower than sequential writing, which leads to the performance degradation of flash memory. Workloads dominated by random writes could wear out the flash chips faster than sequential write-intensive workloads [18]. In addition, compared with sequential writes, random writes will undoubtedly lead to higher garbage collection overhead [19]. Therefore, how to distinguish different data patterns and handle them with different strategies is a key issue. However, the existing data buffer management approaches only can consider spatial or temporal localities for the entire buffer space by LRU-like schemes under NAND flash page or block units, as shown below.

Research on page unit of flash chip has been devised to reduce extra data writes and improve the cache hit rate by taking advantage of the temporal locality of data access [18]. There are several varieties. For example, the clean-first least recently used (CFLRU) policy [20] is able to replace clean pages in advance to avoid the penalty of evicting dirty pages from the buffer area. CFLRU is the basic requirement for a buffer management strategy. Dual locality (DULO) is designed to split the LRU stack into two parts, for spatial and

temporal localities, respectively [21]. Although DULO splits data into refined patterns, its pattern recognition ability is limited by the size of its stack. The probability reference-based LRU (PRLRU) uses the reference probability to select a victim page [22]. To predict the possibility of a page being reaccessed in the future, PRLRU calculates the reference probability of a page by making three variables/parameters. Its limitation lies in how to determine the appropriate parameters for different workloads.

Block granularity or both page and block buffer managements are used to exploit spatial locality and reorganize more sequential writes for flash chips [18]. A flash-aware buffer (FAB) pays more attention to the spatial locality for its representative cache replacement scheme [8]. All the pages cached in the buffer are grouped into block units and conducted by an LRU list. The block padding least recently used (BPLRU) algorithm supervises blocks according to FAB and LRU. Based on these algorithms, it replaces blocks that have not been referenced the longest, that is to say, choosing older blocks as victims [23]. The clean-first dirty-clustered (CFDC) policy combines the CFLRU technique with BPLRU to create two regions, a priority region that manages dirty pages and a working region for loading frequently referenced pages to decide dirty pages [24]. In addition, other schemes are the parallelism-aware buffer (PAB) [25] and garbage collection-aware replacement (GCaR) [26]. As we all know, flash pages in a block are always accessed by different frequencies. However, for block-related granularity buffer management, they are treated as one block unit that will impose extra eviction/fetching loads on the buffer management, especially for random-dominant workloads. Block granularity-based buffer management should be prohibited as its coarse unit.

Previous algorithms consider spatial or temporal localities for the entire buffer space, focusing on the buffer granularity organization, page, block, or page and block units. These conventional policies are all restricted by their working buffer size. That is, the smaller their buffer sizes are, the worse the performance of the buffer strategies is. Therefore, the limitations of existing buffer replacement algorithms should be overcome.

For SSD, not only DRAM management but also FTL and garbage collection can also be optimized by accurate data patterns. The flash translation layer (FTL) plays a key role in the management of NAND flash chips due to erase-before-write feature of flash memory. With the help of FTL software, storage devices composed of NAND flash chips (solid-state driver) can be transparently regarded as ordinary hard disk drive (HDD) devices by existing file systems, with hiding the technical details of NAND flash. FTL receives read and write requests and maps a logical address to a physical address in NAND flash memory. Note that the access patterns (sequential access or random access behavior) of data created by different applications will affect FTL design.

Garbage collection (GC) is also an important process in the performance of SSD. NAND flash memory has relatively long erase times, as ERASE operations are completed one block at a time. With the FTL, this long erase time becomes transparent and it is used to free this invalid memory space so as to allow further PROGRAM operations. In particular, caching-aware garbage collection can dynamically adjust cache space to accommodate garbage collection. For example, a block containing sequential data (data with similar lifetime) will save more overhead compared with a block containing random data (data with enormously different lifetime). Considering data lifetime sorted by our PC technology, caching-aware GC will save SSD resources and benefit SSD performance.

Before introducing our new technique, we use analytical models to find the key performance factors and explain the motivation of our design in Section 3 to show why and where the current works can be improved.

3. Modeling and Motivation

To better understand the buffer management influence qualitatively, we first describe two models based on a normal page-level mapping SSD: a model of storage performance and a write amplification factor model. Then, we present the motivation.

3.1. Analytical Models. Performance Model. In this model, we aim to evaluate the average time of dealing with a page request (read/write). All the read and write requests are separated by their ratios of occurrence. R_w is for write requests, and R_r is for read requests ($R_r + R_w = 1$). A required read or write page is present in the buffer, called an average buffer hit ratio, H_{ar}. For read requests, when a victim entry should be evicted to make space for required entries, the cache eviction penalty is $(P_d * T_{df})$. P_d is the dirty probability during the replacement process. T_{df} equals $T_{\text{write}} + T_{\text{read}}$, where T_{write} denotes dirty pages that should be flushed back flash chips, and T_{read} is the cost of reading target pages from flash. Therefore, the average service time of a read request is as follows:

$$T_{art} = (1 - H_{ar}) * \left\{T_{\text{read}} + \left(P_d * T_{df}\right)\right\}. \quad (1)$$

We can also deduce that the average service time of a read request is as follows:

$$T_{awt} = (1 - H_{ar}) * \left\{T_{\text{write}} + \left(P_d * T_{\text{write}}\right)\right\}. \quad (2)$$

Write Amplification Model. The write amplification model is a good metric to evaluate the extra write problem. Read requests might lead to more extra page writes that are beyond all user page writes, N_{upa}, $*$, R_w. N_{upa} represents the number of user page accesses in a workload. Due to the buffer management target of this article, we only take into consideration the buffer effect, ignoring the garbage collection feature in flash devices. The extra writes will only occur in read requests. If write requests hit in the buffer, write requests can obviously reduce the writing number. N_{piw} is the possible increased extra writes from read operations. $N_{piw} = (1 - H_{ar}) * P_d * N_{upa} * R_r$. N_{prw} is the possible reduced writes from write operations. $N_{prw} = H_{ar} * N_{upa} * R_w$.

Under the condition, $R_w > 0$, the write amplification factor will be as follows:

$$A = \frac{N_{upa} * R_w + N_{piw} - N_{prw}}{N_{upa} * R_w},$$
$$= \frac{N_{upa} * R_w + \left((1 - H_{ar}) * P_d * N_{upa} * R_r\right) - \left(H_{ar} * N_{upa} * R_w\right)}{N_{upa} * R_w}. \quad (3)$$

From (1) and (2), the page service cost can be reduced by either reducing the probability of replacing a dirty entry P_d or increasing the buffer hit rate H_{ar}. From (3), to control the write amplification, we can adopt the same conclusion, increasing H_{ar} or reducing P_d. To increase H_{ar}, better pattern classifications are necessary. For example, preferring to keep looping data (representing that data will be accessed in the future) in a buffer can improve H_{ar} compared with random data. Furthermore, previous buffer studies [16, 27] have shown that pattern-based schemes can achieve better hit ratios than pure recency/frequency schemes. Therefore, distinguishing data patterns and then keeping the most appropriate data in the buffer are clearly effective in buffer management, which are also concerns of our article.

3.2. Motivation. After describing the analytical models, in the motivation part, we compare the merits and shortcomings between URM and conventional LRU-based buffer management policies.

3.2.1. Drawbacks of LRU-Based Policies. For traditional buffer management schemes (BMSs), there are four drawbacks compared with URM, (i) crude data classifications, hot/cold; (ii) restriction of regularity recognition ability by cache size, named locality-regularity recognition in this study; (iii) impaired replacement scheme; and (iv) no reasonable prefetching support.

Crude Data Separation. Those LRU-based schemes and frequency/recency-based schemes can only distinguish high frequency or recent features from the workload. Hence, all of them are oblivious to fine-grained request patterns, such as sequential and looping references. When complex patterns appear at the same time and all data in different patterns are mixed together, it is impossible to distinguish sequential data from random data only by recency or frequency statistics. As we mentioned in Section 3.1, without clear data patterns, a management scheme cannot make the optimum eviction/prefetching decision.

Limited Regularity Recognition. As the pattern identification module of URM (in Section 4.2) is designed in the host layer, it is not affected by the size of the data buffer in the NAND flash storage device. However, the unstable hot recognition ability of LRU is dramatically affected by the storage device's buffer size. By increasing random data, the whole data buffer will be exhausted. Therefore, in these circumstances, any kind of pattern identification policy will lose effectiveness.

Impaired Replacement Scheme and No Reasonable Prefetching Support. As mentioned before, the replacement scheme depends on pattern classification ability. If data patterns cannot be identified, the buffer replacement policy will fail to achieve the best performance. For example, given a sudden appearance of a large number of random data, those young random data (cold data) might replace old but hot data. Furthermore, for prefetching ability, the LRU-based buffer management policies are not able to obtain reasonable prediction parameters to guide when and how long data sequences are prefetched. Only the pattern-based policy can support prefetching according to its sequential or looping pattern identification.

3.2.2. Advantages of PC-Based Management. Fine-grained patterns based on PC URM with fine-grained patterns overcome almost all of the shortcomings above, which can bring about significant benefits. Compared with the traditional hot/cold classification, URM extends the hot patterns into clustered hot and looping patterns and extends the cold patterns into sequential and random patterns in detail. The looping pattern means sequential data occur repeatedly with a regular interval in one PC. The clustered hot pattern means a cluster of data occurs repeatedly in one PC (no strong sequential pattern). The sequential pattern means that sequential data only appear once in one PC. The random pattern means that data have not been detected in patterns similar to the above patterns in one PC.

Global Patterns Detected by PC without Extra Cost in Flash Storage. The PC technique is used to group "from-the-same-program-context" data together in the host layer. During the process of our PC exploration, the key insight is that in most applications, a few dominant I/O activities exist and most dominant I/O activities have definite data access patterns. For instance, the GCC application that builds the Linux kernel uses only one single PC indicator to trigger all header files, and thus, our pattern identification module is able to detect the looping pattern. Once one PC pattern is fixed, the next data coming from the same PC will be directly regarded as a looping pattern. This ensures that our pattern identification module in Section 4.2 only temporarily incurs a cost reflecting a small kernel buffer space, and there is no extra pattern recognition overhead in the NAND flash storage device.

4. Design of URM

In this section, we describe the program counter-technique in the kernel and the RAM management scheme in the NAND flash storage device. The URM is designed in the state of software deployed in the kernel, not in the state of hardware. It is a lightweight design, which only requests less than 10 MB space in kernel part. Figure 1 shows an overview of URM architecture. The URM is designed for RAM management of SSD rather than RAM management of computer. Therefore, on the right side of Figure 1, we draw the architecture of the SSD. The SSD consists of NAND flash chips, RAM, and a microcontroller. In this study, we focused on the role of RAM and how to manage it. This architecture has three modules. The PC calculation module can calculate

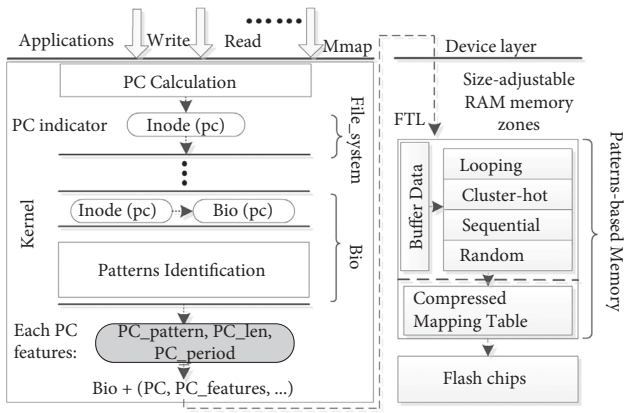

Figure 1: Overall architecture of URM.

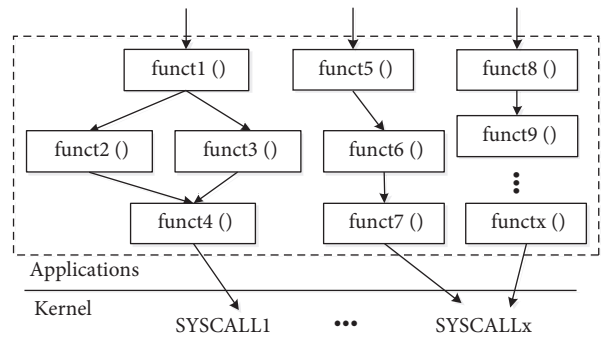

Figure 2: An example of the function calling tree.

a PC indicator for every request. The pattern identification module distinguishes each request's pattern and assigns a pattern to each request's PC indicator. After receiving a request with its corresponding PC indicator from the pattern identification module, the pattern-based memory module starts to deal with this request in the corresponding buffer zone according to this request's pattern.

4.1. PC Calculation. Program contexts have been proven to provide a possible and highly effective means of recording the context of program behavior [16]. In Figure 2, several application examples are exhibited to simply describe a function calling tree of realistic applications. Applications running in the operating system consist of many function call routines, as shown in Figure 2. Every function call routine in this tree is unique, for example, a set of functions, funct4()˜funct2()˜funct1(). All function address values can be searched in the virtual address space of the operating system. Therefore, each routine can be uniquely represented by the sum of all the function addresses from the target routine, called a PC indicator.

As a PC indicator uniquely represents a task (function routine) and rather constant behavior, and determining a PC indicator for every routine invoked by Syscalls is the task of the PC calculation module. However, since full scanning in the virtual address space for every function routine is almost an unbearable overhead, the PC calculation module needs to find an appropriate number of function candidates. Based on our empirical tests, approximately 5 address candidates are enough to identify different calling routines and provide enough time savings. Moreover, under the current high CPU frequency and our lightweight architecture, this overhead can be further reduced [17].

4.2. Pattern Identification. Now, all I/O requests have been tagged with PC indicators by the pattern identification module. The next task is for the pattern identification module to distinguish the patterns for the coming requests. In theory, if one routine (represented by a PC indicator) repeatedly creates I/O requests, this routine's PC indicator will be treated as a hot pattern in the pattern identification module.

We explain four kinds of data patterns here. For the looping pattern, providing a succession of requests invoked by the same PC indicator occurs repeatedly, and this PC indicator can be regarded as a looping PC. Generally, a request is composed of many file pages. Then, data in a looping pattern require that all pages are consecutive as well. Compared with the looping pattern, file pages in a sequential pattern only need to meet the sequential requirement, not the repeated access requirement. In terms of the clustered hot pattern, it can be treated as a small-scale looping pattern. There are three parameters describing each pattern: PC_pattern to show pattern categories, PC_len to describe this PC indicator's sequential page numbers, and PC_period to denote period values of looping and clustered hot patterns. With the PC_len value, URM could finish the prefetching operation in the buffer management. PC_period is also a parameter used in buffer management; for example, data represented by shorter period value PCs are more suitable to be kept in buffer than data in longer period value PCs. Finally, except for the three patterns above, the remaining PC indicators will be classified into random patterns, which have no clear sequential or repeated features.

In this design, the PC_len parameter represents preloading mapping information, which might be accessed in the future. At first, the PC_len parameter is worked out by averaging sequential length from requests indicated by a specified PC value. However, the average value is not precise. For example, when the shortest sequential request comes, the predicted average length might be much longer than the reasonable preloading length. Aggressive prefetching could lead to unnecessary loadings and replacements, decreasing the prefetching benefit. Because the unnecessary mapping information will evict mapping information, which may be used, the preloading action will also aggravate mapping dealing overhead.

As for the limitation of NAND flash, in our design, a translation page is supposed to be composed of 1024 mapping entries. Therefore, we plan to partition these 1024 mapping entries into 10 even regions. Each region contains about 102 entries. For a coming sequential or looping request, the URM predicts which part among 10 even sections is suitable for prefetching length prediction result. Compared with the average strategy mentioned earlier, the fine-grained prediction is more accurate.

Now, we propose a new strategy, the multivariate classification method. Our multivariate classification is able to be transformed into multiple binary classifications based on logistic regressions. For each logistic regression strategy, we set three corresponding parameters. The first parameter (PC) is the PC value mentioned earlier, because each request has a corresponding PC. The second parameter (LBN) is the start LBN value of this request. The third parameter (AV) is the average preference length of its appointed PC (PC_len parameter). There are ten binary classification equations listed. $g_0(x) = W_{0a} + W_{0b}PC + W_{0c}LBN + W_{0\,d}AV$; $g_1(x) = W_{1a} + W_{1b}PC + W_{1c}LBN + W_{1\,d}AV$; and $g_9(x) = W_{9a} + W_{9b}PC + W_{9c}LBN + W_{9\,d}AV$.

Then, the logistic regression equation will be shown as follows:

$$g_i(x) = W_{ia} + W_{ib}PC + W_{ic}LBN + W_{id}AV.i \in [0, 9]. \quad (4)$$

The sigmoid equation will be as follows:

$$\text{sigmoid}\left(g_i(x)\right) = \frac{1}{1 + \exp^{-g_i(x)}}. \quad (5)$$

All the parameters from W_{ia} to $W_{i\,d}$ can be trained according to our collected workload sample. After fixing the parameters, we can work out ten sigmoid values for every coming request. Then, the largest sigmoid value is our prediction result.

$$\text{Result}_{\max} = \text{MAX}\left(\text{sigmoid}\left(g_i(x)\right)\right).i \in [0, 9]. \quad (6)$$

For example, for a coming request, we assume that when i variable equals 3, $Result_{\max}$ achieves maximum value. Our predicted prefetching length for this coming request is 408 ($408 = 102 * 4$). Finally, compared with the average method, the prediction accuracy of prefetching length is improved by 15%.

The program counter-technique utilization is not a new area of research. For example, Gniady et al. proposed UBM [28] and Kim et al. have proposed buffer classification based on a program counter [16]. However, the existing program counter-technique designs use the physical block number (PBN) to train PC patterns, which will result in the following inevitable problems. For example, in those aging disks, files have increasingly more "wholes" (PBN wholes in the file system layer), which destroy the sequential features of PBNs. These modified PBNs will change the old PC pattern and then destroy the effectiveness of the PC technique. In URM, we replace PBNs with the logical page numbers in a file so that we can maintain the PC pattern without the trouble from data "wholes."

4.3. Pattern-Based Memory Organization. In Figure 1, FTL in the flash storage device can receive a bio-request along with its PC pattern and the pattern's parameters from the pattern identification module. According to these predicted parameters, the data buffer can make the decision to evict (accommodating coming write requests) or preload future data (for read requests) from the flash chip. The data buffer space and mapping cache space in the RAM chip of a NAND flash storage device are plotted in detail in Figures 3 and 4, respectively.

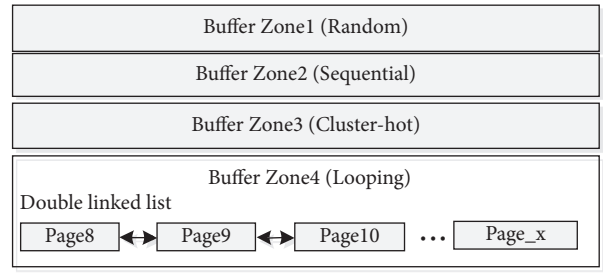

Figure 3: Four data buffer zones.

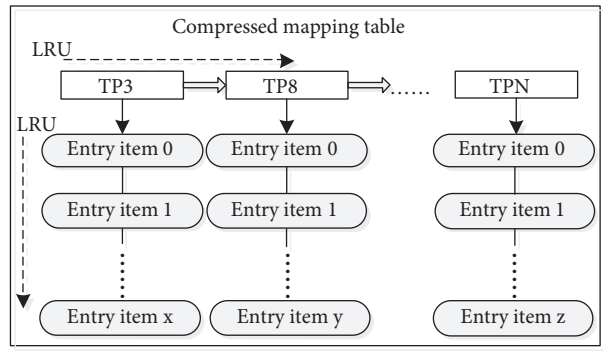

Figure 4: Compressed mapping table.

Figure 3 shows four size-adjustable zones in the data buffer. In this figure, we unfold the looping pattern zone and fold the other three zones because they are organized in the same data structure, the double-linked list. The detailed buffer management scheme is shown in Section 4.4.

Figure 4 exhibits how the mapping table is organized, and Figure 5 shows how to compress and decompress every entry item in Figure 4. In the compressed mapping table, TPN represents one translation page, which is filled with physical page numbers (PPNs). For instance, assuming that one PPN costs 4 bytes, one 4 KB flash page can save 1024 PPNs. In this study, the entry item is redesigned similar to other compressed mapping table papers [29, 30]. After compression, the mapping table is organized by the double-linked LRU list.

The entry item example (5,2,5,D) in Figure 5 has been split into 4 parts, relative logical page number (RLPN) offset, PPN offset, consecutive length, and dirty flag. After decompression, one compressed entry item contains five normal mapping pairs, indicating the mapping table size reduction. Generally, LPN is one logical page number in the storage device scope. However, in this study, the relative LPN (RLPN) belongs to a TPN scope, and every TPN in our design has a maximum of 1024 mapping pairs. Therefore, the size of 10 bits for the RLPN part in the entry item is sufficient. The other three parts in the entry item cost 4 bytes, 5 bits, and 1 bit, respectively, in this article. The continuous length of 5 bits can accommodate at least 32 consecutive mapping pairs, which cost 256 ($32*8$) bytes. The maximum compression ratio is approximately 43 times. Finally, for the entry item, its total of 6 bytes are flexible enough to reduce

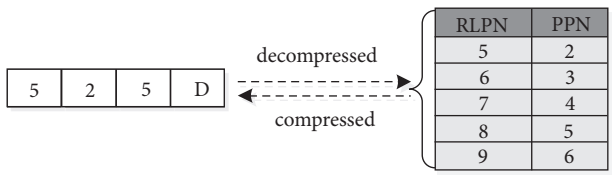

Figure 5: Compression of an entry item.

the mapping cache space and large enough to store consecutive compressed mapping pairs.

4.4. Pattern-Based Schemes of URM. All the data structures have been introduced above. The key innovative points in our design are the new buffer replacement policy, prefetching method, and automatic size adjustment between the mapping cache and data buffer.

Before illustrating the replacement policy, we define the buffer hit when the data are present in the data buffer. In contrast, a buffer miss indicates that there are no target data in the buffer. Assuming there is not enough buffer space for accommodating data, a reasonable eviction strategy can help reduce the flush back cost and significantly improve device performance. Inspired by pattern definitions and other pattern-based cache research [28], we sort the eviction priority sequence as "sequential > random > cluster-hot > looping." The reason is clear from our qualitative analysis. Sequential pattern data only require access once, which is completely useless in the future. Random pattern data possess an irregular repetition rate. Clustered hot pattern and looping pattern data are definitely accessed again in the future. However, the continuous length of looping pattern data is longer than that of clustered hot pattern data, which has the highest priority.

The prefetching method depends on our predicted parameters. For the read operation, provided that the read pattern is looping, sequential, or even clustered hot, the parameter PC_len denoting the possible length can be used to implement preloading. The entire RAM size is fixed and controlled by the FTL. The compressed mapping table size is dynamic. Compared with the normal page mapping table, the compressed mapping table saves more RAM space. Then, our automatic size-adjustable scheme helps URM to share the saved RAM space with the data buffer. Moreover, with the pattern recognition ability, we can also improve the compression scheme well. For example, data in the random pattern are not eligible for compression due to their poor compression ratio. Compared with looping or sequential data, compression for random data is an ineffective and time-consuming process. Traditional buffer schemes cannot support the elaborate management provided by URM.

At last, there are several steps to reduce the time overhead of our URM. First of all, we make use of the computing power of the CPU to improve the performance of SSD device, which will not cost any time of SSD. Secondly, in the step of working out PC values, we use 5 address candidates to replace the full-scale search in pattern recognition. At last, the two-level (double-linked) list data structure has been used to benefit fast searching.

4.5. Basic Operations in URM. This section describes the process of dealing with upcoming requests and their PC indicators and parameters by explaining the basic operations and writing and reading processes.

When write operations arrive, FTL needs to find room to accommodate them. If the available buffer space is large enough, FTL will directly store arriving write requests. In URM, the choice of the zone depends on their PC indicators. Assuming a write request has been assigned to a looping pattern, it should be stored in a looping zone. Once the data buffer no longer has enough space to store arriving request data, according to the eviction policy described above, the victim buffer data should be flushed back into flash chips if the victim data are dirty, and the FTL should update the mapping table at the same time. However, for clean victim data, FTL only needs to delete clean data from the data buffer.

In terms of searching the mapping table, the first step is to locate the translation page number (TPN) as shown in Figure 4. We use an example to explain the search process. Given a visited page with a page number of 2055, we can determine that its TPN number is TP1 (2055/2048 =1) and RLPN is 6. By searching 6 RLPNs in TP1, we can obtain an entry item. Then, the PPN address can be found in the entry item. Updating or modification of the compressed mapping table is a new operation in URM, which might split an old entry item into two or three new entry items since updated PPNs have destroyed the consecutive characteristics of old PPNs.

When a NAND flash storage device services a read request, FTL first searches in the data buffer before reading it from a flash chip. Note that the data buffer search process is not in a specific zone based on our predicted pattern, but in all four data zones. We cannot ensure that the arriving data stay at their predicted zone due to the complicated applications. Therefore, once the read data appear in a zone that is different from our predicted pattern, PC pattern, it is necessary to migrate the data from the old zone to our predicted zone, which is called zone data movement. Given the buffer misses occurring in the data buffer, FTL starts to acquire data from flash memory chips after searching the mapping table to locate corresponding PPNs. Before fetching, the data buffer eviction is the same as that of write operations. Through the eviction policy, FTL retains suitable hot data and expels cold data. This reading process also invokes the prefetching policy described in Section 4.4. All the eviction and prefetching performances are exhibited in Section 5.

5. Evaluation

5.1. Experimental Design. We have implemented kernel modules, PC calculation, and pattern identification modules, mentioned in Figure 1 in the Linux kernel 3.16 and the pattern-based memory module in a trace-driven SSD simulator named FlashDriver [31]. The SSD simulator is configured with a capacity of 32 GB, and each chip contains 256 blocks, each of which contains 128 pages (4 KB). The total RAM is set as 33 MB to 48 MB. Note that, for a capacity of 32 GB, the uncompressed pure page mapping cache size is

Table 1: Specification of workloads.

Workloads	Apps run concurrently	#.of requests	Read ratio	Avg. req. size
Multi1	cscope, fio-random	304,109	89%	6.2 KB
Multi2	glimpse, fio-random	280,016	70%	5.4 KB
Multi3	cscope, glimpse, GCC, fio-random	603,321	68%	5.76 KB
Multi4	RocksDB, fio-random	582,321	30%	7.3 KB

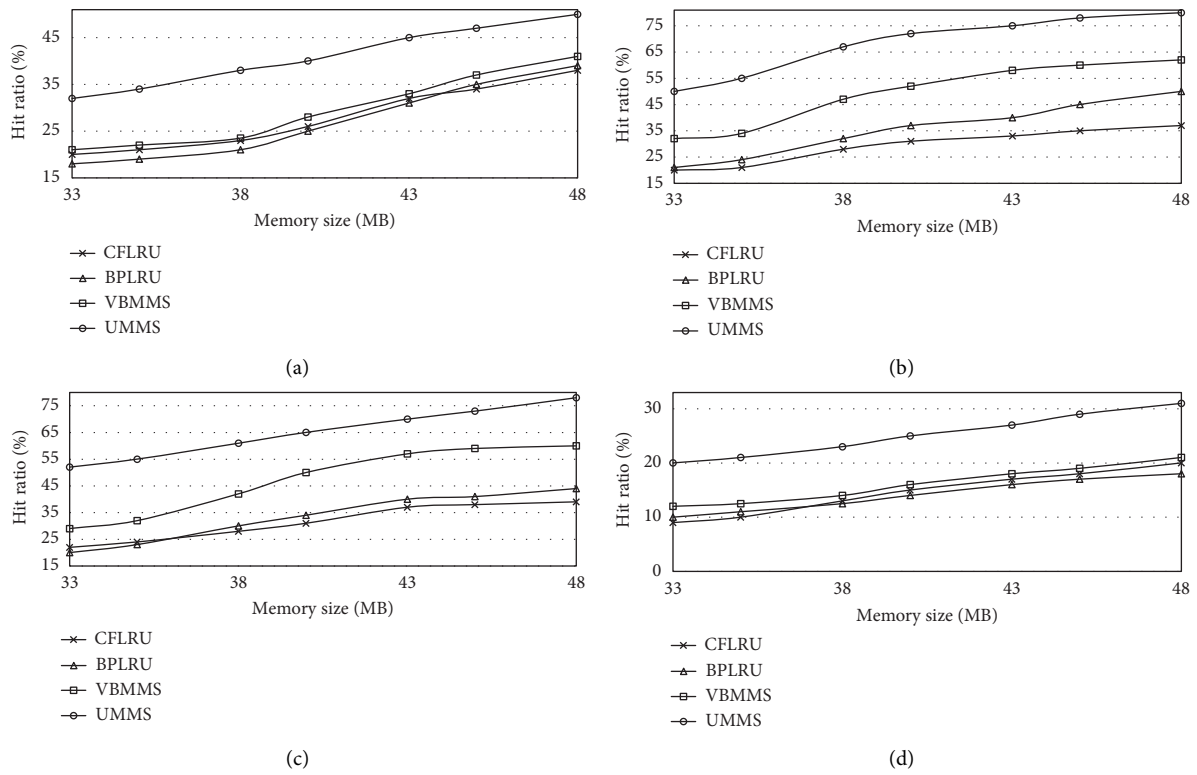

Figure 6: Buffer hit ratio comparison. (a) Multi1 workload. (b) Multi2 workload. (c) Multi3 workload. (d) Multi4 workload.

32 MB. The basic flash memory chip's page reading and page programming timing parameters are set to 25 μs and 200 μs, respectively. To carry out an objective evaluation, we compare URM to several classical BMSs, CFLRU, BPLRU, and VBBMS.

Since URM obtains pattern predictions from the host, we need to collect workloads from realistic applications. Therefore, several well-known applications have been selected, cscope, glimpse, GCC, and RocksDB. cscope is an examination tool used to check source code. It creates looping patterns and other patterns. glimpse is a retrieval tool for text files. It consists of many sequential and looping patterns. GCC is the GNU compiler collection. It is composed of sequential and other patterns. RocksDB is an excellent persistent key value database for fast storage. Its patterns depend on the test benchmarks.

To simulate realistic environments, a random application fio-random has been executed concurrently to mix simple sequential or looping patterns, where that random and other strongly regular patterns have been shuffled together. fio-random is the "fio" application issued with random reads and writes under the configuration of "sync engine, direct io, and bs = 4 KB."

In our evaluations, we choose four multiple application workloads. They are denoted by Multi1, Multi2, Multi3, and Multi4. Their characteristics are shown in Table 1. Multi1 contains more than 60 percent of fio-random data. Multi2 is dominated by looping and sequential patterns, with less than 30 percent of fio-random data. Multi3 is composed of a balance of four kinds of patterns. Multi4 is dominated by sequential write data.

5.2. Performance Comparisons. In this section, we choose two metrics, buffer hit ratio and average response time, to prove the effectiveness of URM. We compare URM with three existing conventional buffer strategies. In CFLRU, its clean-first section ratio is set to 0.5 with a static window size. In BPLRU, we implement a double-linked indexing technique to benefit fast searching. For the VBBMS, its ratio between RRSR and SRSR is fixed as 1 : 1.

Buffer hit ratio (Figure 6) shows the buffer hit ratio comparisons of four workloads when the RAM size varies from 33 MB to 48 MB. For the pure page mapping management, the RAM consists of 32 MB of mapping cache. There are several conclusions. (i) By increasing the buffer

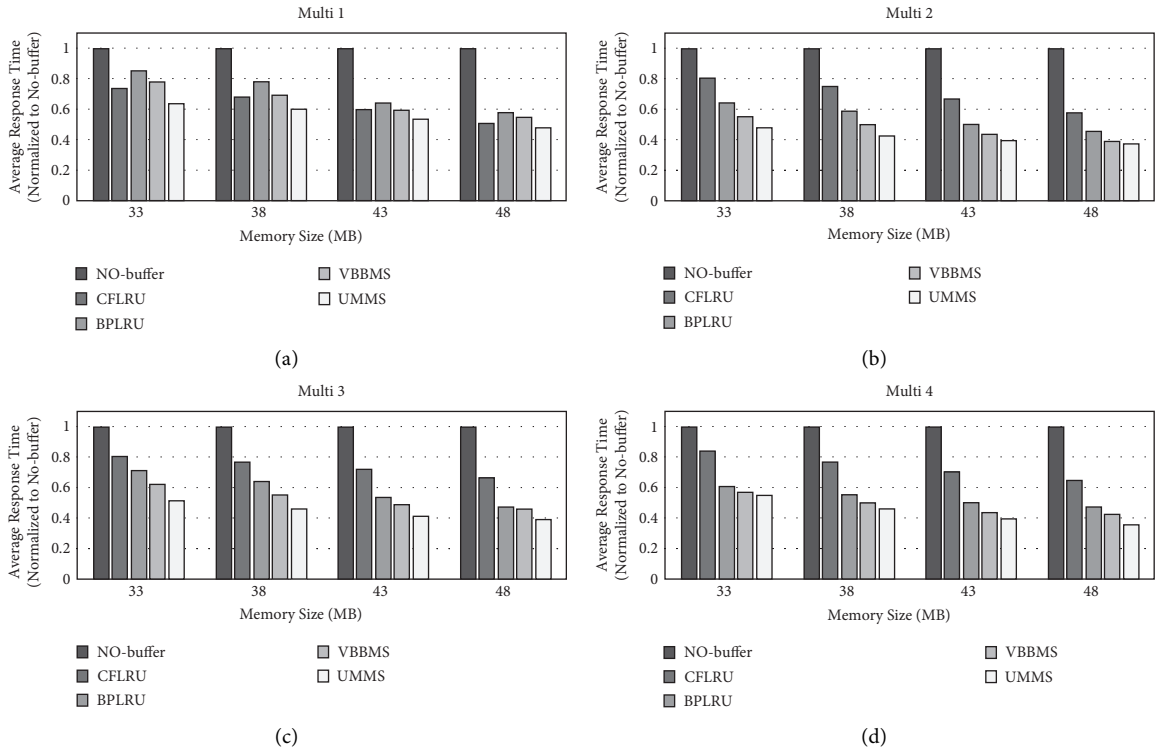

Figure 7: Average response time comparisons. (a) Multi1 workload. (b) Multi2 workload. (c) Multi3 workload. (d) Multi4 workload.

size, all buffer strategies perform better in hit ratios. (ii) If the particular buffer size is not large enough, compared with the page granularity buffer policy (i.e., CFLRU), the block granularity-based buffer policy (i.e., BPLRU) performance will be restricted. Except for Multi4, this conclusion can be observed at all 33 MB points in Figure 6. The reason is that the coarse unit is not suitable for a small buffer size. The Multi4 workload is full of sequential write requests such that BPLRU with block granularity can make use of the continuous features in Multi4. (iii) Although VBBMS is designed to distinguish random and sequential patterns separately, it is not enough to deal with more complex mixed data patterns. Those looping or cluster patterns are new features to VBBMS. That is why VBBMS cannot work better than URM in the buffer hit ratio. (iv) The mapping cache size reduction and fine-grained pattern identification capability are two reasons that permit URM to work best compared with other policies, which can be found in all workload test results shown in Figure 6. Finally, we obtain concrete differences in Figure 6(b), and under a memory size of 43 MB RAM, URM achieves approximately the best improvement, 44%, 34%, and 19%, compared with CFLRU, BPLRU, and VBBMS, respectively. From the perspective of URM, compared with other policies, the overall average hit ratio improvement is 32%.

Average response time (Figure 7) shows the average response time comparisons of the same workloads and the same RAM size scope (from 33 MB to 48 MB). According to the paper [29], the request-response time depends on the buffer hit ratio, the probability of replacing a dirty page, the page flush back cost, etc., not only the buffer hit ratio. For example, the performance comparison shown in Figure 7(b) has proven that the response time gaps are different from the buffer hit ratio gaps in Figure 6(b) under the same RAM size. The reason is that Multi2 with looping and sequential patterns not only improves the hit ratio but also reduces flushing back frequencies of dirty pages. We can also observe that URM is superior to other strategies in terms of response time costs in Figure 7(c), where under the RAM size of 43 MB, URM can reduce time costs by 31%, 12%, and 8% at most compared with CFLRU, BPLRU, and VBBMS, respectively. Compared with other schemes, the total average reduction in response time of URM is approximately 18%.

6. Conclusion

In this study, we devise a novel RAM (buffer and cache) management algorithm called URM, based on the program counter-technique in the host. Its optimal pattern identification ability and automatic size-adjustable ability help URM outperform traditional buffer management strategies in both hit ratio and request-response time indicators. In addition, the mapping size reduction technique permits FTL to save precious memory space in the NAND flash storage device. In the future, it is our goal to accelerate the mapping table decompression/compression process by a hardware approach.

Data Availability

The experimental data used to support the testing of this study have not been made available because of the requirement of our funding.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This research was funded by the Advanced Research Project on Information System Equipment for the PLA during the 13th Five-Year Plan Period (No. 31511030103).

References

[1] W. Feng and N. Deng, "Study on Cell Shape in 3D NAND Flash Memory," in *Proceedings of the 2015 IEEE International Conference on Electron Devices and Solid-State Circuits (EDSSC)*, pp. 387–390, Singapore, June 2015.

[2] L. Tang, N. Kuang, Y. Guo, and Y. Liu, "The development status and future prospects of information processing microsystem," *Microelectronics & Computer*, vol. 38, 2021.

[3] R. Mativenga, P. Hamandawana, T.-S. Chung, and K. Jongik, "FTRM: a cache-based fault tolerant recovery mechanism for multi-channel flash devices," *Electronics*, vol. 9, no. 10, 2020.

[4] G. Oh and S. Ahn, "Implementation of memory efficient flash translation layer for open-channel SSDs," *Hepatology*, vol. 10, no. 1, pp. 142–150, 2021.

[5] N. Megiddo and D. S. Modha, "ARC: a self-tuning, low overhead replacement cache," *FAST*, vol. 3, pp. 115–130, 2003.

[6] B. S. Gill and D. S. Modha, "WOW: Wise Ordering for Writes-Combining Spatial and Temporal Locality in Non-volatile Caches," in *Proceedings of the FAST '05: 4th USENIX Conference on File and Storage Technologies*, San Francisco, CA, USA, December 2005.

[7] B. Debnath, S. Subramanya, D. Du, and D. J. Lilja, "Large block CLOCK (LB-CLOCK): a write caching algorithm for solid state disks," in *Proceedings of the 2009 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pp. 1–9, London, UK, September 2009.

[8] H. Heeseung Jo, J.-U. Jeong-Uk Kang, S.-Y. Seon-Yeong Park, J.-S. Jin-Soo Kim, and J. Joonwon Lee, "FAB: flash-aware buffer management policy for portable media players," *IEEE Transactions on Consumer Electronics*, vol. 52, no. 2, pp. 485–493, 2006.

[9] C. Du, Y. Yao, J. Zhou, and X. Xu, "VBBMS: a novel buffer management strategy for NAND flash storage devices," *IEEE Transactions on Consumer Electronics*, vol. 65, no. 2, pp. 134–141, 2019.

[10] J. Liu, Y. Lan, J. Liang et al., "An efficient schema for cloud systems based on SSD cache technology," *Mathematical Problems in Engineering*, vol. 2013, Article ID 109781, 9 pages, 2013.

[11] Y. Smaragdakis, S. Kaplan, and P. Wilson, "Eelru," *ACM SIGMETRICS - Performance Evaluation Review*, vol. 27, no. 1, pp. 122–133, 1999.

[12] F. Cao, Z. Yang, J. Ren et al., "Sparse representation-based augmented multinomial logistic extreme learning machine with weighted composite features for spectral–spatial classification of hyperspectral images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 6556, no. 11, pp. 6263–6279, 2018.

[13] M. D. Powell, A. Agarwal, T. N. Vijaykumar, B. Falsafi, and K. Roy, "Study on cell shape in 3D NAND flash memory," in *Proceedings of the 34th ACM/IEEE International Symposium on Microarchitecture. MICRO-34*, pp. 54–65, Singapore, June 2001.

[14] A.-C. Lai and B. Falsafi, "Selective, accurate, and timely self-invalidation using last-touch prediction," in *Proceedings of the 27th International Symposium on Computer Architecture (IEEE Cat. No. RS00201)*, pp. 139–148, Vancouver, BC, Canada, June 2000.

[15] X. Li, Z. Zhai, and X. Ye, "AOIO: application oriented I/O optimization for buffer management," *Symmetry*, vol. 13, no. 4, 2021.

[16] C. Gniady, A. R. Butt, and Y. C. Hu, "Program-counter-based pattern classification in buffer caching," *Osdi*, vol. 4, 2004.

[17] T. Kim, D. Hong, S. S. Hahn et al., "Fully automatic stream management for multi-streamed ssds using program contexts," in *Proceedings of the 17th USENIX Conference on File and Storage Technologies FAST 19*, vol. 54, pp. 295–308, Boston, MA, USA, February 2019.

[18] Q. Wei, C. Chen, and J. Yang, "CBM: A Cooperative Buffer Management for SSD," in *Proceedings of the 2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–12, Santa Clara, CA, USA, June 2014.

[19] S.-J. Chae, R. Mativenga, J.-Y. Paik, M. Attique, and T.-S. Chung, "DSFTL: an efficient FTL for flash memory based storage systems," *Electronics*, vol. 9, no. 1, Article ID 20205, 2020.

[20] S.-yeong Park, D. Jung, J.-uk Kang, J.-soo Kim, and J. Lee, "CFLRU: a replacement algorithm for flash memory," in *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pp. 234–241, Seoul, Korea, October 2006.

[21] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: an effective buffer cache management scheme to exploit both temporal and spatial locality," in *Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, vol. 4, Francisco, CA, USA, December 2005.

[22] 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.

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

[24] Yi Ou, T. Härder, and P. Jin, "CFDC: a flash-aware replacement policy for database buffer management," in *Proceedings of the 5th International Workshop on Data Management on New Hardware*, vol. 12, pp. 15–20, Rhode Island, USA, June 2009.

[25] X. Guo, J. Tan, and Y. Wang, "PAB: parallelism-aware buffer management scheme for NAND-based SSDs," in *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pp. 101–110, San Francisco, CA, USA, August 2013.

[26] S. Wu, Y. Lin, B. Mao, and H. Jiang, "GCaR: garbage collection aware cache management with improved performance for flash-based SSDs," in *Proceedings of the 2016 International Conference on Supercomputing*, pp. 1–12, Istanbul, Turkey, June 2016.

[27] J. Choi, S. H. Noh, S. L. Min, and Y. Cho, "Towards application/file-level characterization of block references: a case for fine-grained buffer management," in *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 286–295, Santa Clara, CA, USA, June 2000.

[28] J. M. Kim, J. Choi, J. Kim et al., "A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references," in *Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, vol. 4, San Diego, CA, USA, October 2000.

[29] Y. Zhou, F. Wu, P. Huang, X. He, C. Xie, and J. Zhou, "Understanding and alleviating the impact of the flash address translation on solid state devices," *ACM Transactions on Storage*, vol. 13, no. 2, pp. 1–29, 2017.

[30] Z. Qin, Yi Wang, D. Liu, and Z. Shao, "A two-level caching mechanism for demand-based page-level address mapping in NAND flash memory storage systems," in *Proceedings of the 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*, pp. 157–166, Chicago, IL, USA, April 2011.

[31] Dgist-datalab, "Dgist-datalab," FlashDriver, 2021, https://github.com/dgist-datalab/FlashDriver.