LCR: Load-aware Cache Replacement Algorithm for Flash-based SSDs

Caiyin Liu[1], Min Lv[1], Yubiao Pan[2], Hao Chen[1], Yongkun Li[1], Cheng Li[1], Yinlong Xu[1]
[1]School of Computer Science and Technology, University of Science and Technology of China
[2]College of Computer Science and Technology, Huaqiao University
{cyliu99,cighao}@mail.ustc.edu.cn, {ylxu, ykli,lvmin05,chengli7}@ustc.edu.cn, panyubiao@hqu.edu.cn

***Abstract*—Flash-based SSDs are usually equipped with an on-board cache to further improve system performance by smoothing the gap between the upper-level applications and lower-level flash chips. Since modern SSDs are usually composed of multiple flash chips, and the load of flash chips are significantly different, it is very meaningful to be aware of the chip load condition when designing a cache replacement algorithm. Nevertheless, existing cache replacement algorithms only consider to reduce the cache miss ratio so as to reduce the I/O requests to the underlying flash memory as much as possible, none of them considers the load condition of flash chips. In this paper, we propose a Load-aware Cache Replacement algorithm, called LCR, to improve the performance of flash-based SSDs. The basic idea is to give a higher priority to cache the blocks on overloaded flash chips. We evaluate the performance of our scheme by using a trace-driven simulator with multiple real-world workloads, and results show that compared with the most common algorithm LRU and the state-of-the-art algorithm GCaR, LCR reduces the average response time by as much as 39.2% and 12.3%, respectively**

***Index Terms*—Flash; SSD; Cache Replacement**

I. Introduction

NAND-flash based Solid-State Disks (SSDs) show superior performance over hard disks in lower access latency, smaller size, lower energy consumption, no noise, and shock resistance [6]. In recent years, SSDs have received a great deal of attention from industry and academia. Besides the deployment on personal computers and mobile devices, they also have been widely deployed in the high performance computing and enterprise environments [2, 7]. Moreover, with the drop of its per-bit cost, SSDs will be more widely deployed.

In order to reduce the access latency, SSDs usually adopt an on-board device cache, such as DRAM or SDRAM, which is used for smoothing user I/O requests traffic [22]. The user I/O requests from upper-level file systems can be kept in the cache for as long time as possible before being flushed to the flash chips. In other words, the on-board cache plays an important role in offering a better I/O performance since a majority of requests can be serviced at the cache speed [20]. However, the effectiveness of the cache mechanism relies on the cache replacement policy [10].

Due to its simple and effective exploitation of temporal locality, the Least Recently Used (LRU) is the most widely used cache replacement algorithm [4]. Over the years, a large number of cache replacement algorithms have been proposed, such as 2Q [14], MQ [26], LIRS [12], ARC [18], DULO [11], etc. These traditional cache replacement algorithms have been optimized in various ways for operating systems, but they are customized for disk-based secondary storage. Unfortunately, flash-based SSDs exhibit unique characteristics compared to Hard Disk Drives (HDDs), such as asymmetric read and write latencies [13], out-of-place update policy, and Garbage Collection (GC) to reclaim invalid pages and to create free space for incoming write requests. Due to the huge overhead of GC process [25] and the evidently unbalanced distribution of I/O requests on different flash chips, it is necessary to revisit various cache replacement policies for disk-based storage, so as to optimize them for flash-based SSDs.

There are also multiple studies on cache replacement algorithms for flash-based SSDs, and they optimize the cache performance from different perspectives. Firstly, CFLRU [19] and LRU-WSR [15] are designed to reduce the average replacement cost by making an effort to minimize the number of write requests from upper-level file system to the backend devices. Secondly, GC-ARM [9] and PUD-LRU [8] aim for improving the GC efficiency by considering how to reorganize data blocks so as to improve the process of flushing data to flash chips. Finally, GCaR [24], which is aimed at maximally reducing the contentions between the user I/O operations and the GC-induced I/O operations. The basic idea of the GCaR is giving a higher priority to cache the data blocks belonging to the flash chips that are currently doing GC.

However, none of existing works take the load of flash chips coming from real-time workloads into consideration, making them ineffective in reducing the expensive I/O traffic to overloaded flash chips. We find that alleviating the contention and interference between the flushed I/O requests and the total I/O operations on overloaded chips will greatly reduce both the tail and average response time of user I/O requests.

In this paper, we propose a Load-aware Cache Replacement algorithm, named LCR, to improve the performance of flash-based SSDs. The basic idea is to give a higher priority to cache the data blocks on the flash chips that are in overloaded state. The main contributions of this paper are as follows.

- We propose a Load-aware Cache Replacement algorithm, named LCR, which selects the victim data block in the cache line according to the load levels on flash chips and assigns a higher priority to cache the data blocks on overloaded chips.
- We propose a two-level caching policy by dividing the LRU list into two regions, working region and destaging

978-1-5386-8367-5/18/$31.00 ©2018 IEEE

region. The working region caches the recently accessed data blocks, which are the main source of cache hits. The destaging region keeps the data blocks which are candidates for eviction.

- We propose a scheme to trade off the miss ratio and the miss penalty by narrowing down the range of victim blocks to be selected in the destaging region. Meanwhile, it can also reduce the computation cost to monitor the workloads on all flash chips.
- We conduct extensive trace-driven performance evaluations based on Disksim simulator with SSD extension. The results show that LCR reduces the average response time over the most common and most recent algorithms, LRU and GCaR, by as much as 39.2% and 12.3%, respectively.

The rest of this paper is organized as follows. We introduce the background and motivation of our work in Section II. We then present the design details of LCR and its performance evaluation in Section III and Section IV, respectively. Finally, we conclude this paper in Section V.

II. Background and Motivation

In this section, we first introduce some background about SSDs, then introduce the state-of-the-art cache replacement schemes for SSDs, and finally motivate this work by discussing how we can take unbalanced loads on different flash chips into consideration in the design of cache replacement algorithms.

A. Flash-based SSDs

As illustrated in Fig. 1 [24], a typical SSD is mainly made up of three components, Host Interface Logic (HIL), SSD controller and NAND flash chips. HIL is responsible for translating host system requests to NAND flash aware requests and vice versa. SSD controller takes charge of executing read, write and erase operations to NAND flash chips according to the requests from HIL.

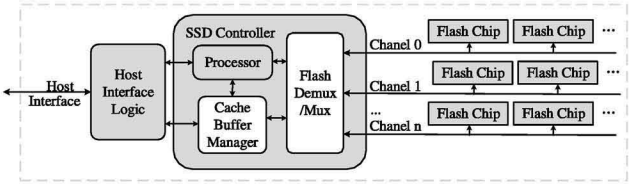


Fig. 1. Logical architecture of a modern SSD.

An SSD is usually composed of multiple flash chips and data are distributed to all chips. So user's I/O requests are distributed on different chips, and they can be executed in parallel. Due to the spatial locality and temporal locality, it is common that applications frequently access the same flash chips at a short time interval [5]. In addition, if a flash chip is frequently accessed, the number of free blocks on the chip may become lower than a predetermined threshold, which triggers frequent GC operations. As a result, if a chip is overloaded, it will induce long waiting time for the I/O requests queued in this chip and postpone the I/O requests to process. In practical systems, the workloads on different chips may be seriously skewed, which is validated in many application scenarios and also in our evaluations as shown in Fig. 2.

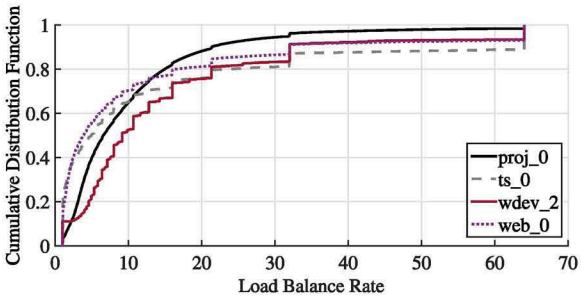


Fig. 2. The load balance rates of different realistic traces.

TABLE I
The Parameters and Operations of SSD

Operations	Access Time	Operation Unit	Typical Size
Read	25μs	Page	4KB
Write	200μs	Page	4KB
Erase	2ms	Block	256KB

The workload on a SSD comes from two sources, the I/O requests from application requests and the ones induced by Garbage Collection (GC). There are three types of workloads on a chip, i.e., read, write and erase. It is well-known that SSDs have asymmetric operation latencies. The read and write operations are performed in pages, however, the erase operation is performed in blocks. Due to the limitation of the erase-before-write updating policy, read is about several times faster than write, and especially, 2-3 orders of magnitude faster than erase. One particular example of the timing parameters of SSDs is shown in Table I [23].

To validate the unbalanced workloads on flash chips. We conduct evaluations on the trace-driven simulator with four realistic traces. Before the execution of our evaluations, the SSD is filled with data such that GC may be triggered during the execution of the traces. We define the load balance rate as the ratio of the maximum load on a flash chip to the average load on all flash chips [21]. The larger the load balance rate is, the more severe the workloads on different flash chips are unbalanced. Fig. 2 shows the load balance rate of different realistic traces. We can see that there are almost 40% of the cases in which the balance rates are larger than 10, which means that the load on the most overloaded chip is more than ten times heavier than average. Moreover, previous studies also have similar findings [16, 17]. All of these studies have revealed that the load conditions on the flash chips are extremely unbalanced and it has a significant impact on the system performance. Although there are many algorithms for load balance on flash-based SSDs, they are only aware of the I/O waiting queue of requests from application layer and balance users' requests. They don't aware of the I/Os comes from the management of flash chips, such as garbage collection

and wear leveling, etc. As a result, the unbalanced load of flash chips remains, especially when garbage collection and wear leveling are performed.

B. The Analysis of Factors Affecting Cache Efficiency

There are many kinds of storage devices, such as RAM, NVM, disk and flash, which show different performance characteristics. The cache mechanism plays an important role in smoothing performance gap between two neighboring layers of the storage devices. Wu et al. [24] proposed a model to analyze the cache efficiency. In details, the Average Memory Access Time (AMAT) is regarded as the most important standard to measure the efficiency of the cache management. AMAT is formulated as follows.

$$AMAT = Hit\_Time + Miss\_Rate \times Miss\_Penalty. \quad (1)$$

As we can see, the AMAT depends on three factors, *Hit_Time*, *Miss_Rate* and *Miss_Penalty*. The *Hit_Time* of internal cache built-in SSDs is consistent. The proposed optimization of cache management makes an effort to reduce the *Miss_Rate* by exploiting access locality to delay the process of flushing upper-level user I/O requests to the low-level flash chips. For *Miss_Penalty*, they assume that the access time of flash chips are almost the same, meaning that the *Miss_Penalty* is a constant. However, the assumption is beneficial to HDDs, but it may not be applicable to SSDs. From aforementioned discussion, we know that the delay of accessing a flash chip is not a constant and depends on its load level. So the *Miss_Penalty* of accessing different chips must be different, and it may vary greatly under seriously skewed workload.

The existing cache replacement algorithms do not take the load levels on flash chips into consideration, e.g., the LRU scheme always evicts the tail data blocks in the LRU list. But if we evict a data block near the tail which is on a chip with lighter load, we may keep almost the same cache miss ratio, but the *Miss_Penalty* will become smaller. So the cache efficiency will be improved. Therefore, both *Miss_Rate* and *Miss_Penalty* should be taken into consideration when designing cache replacement algorithms for flash-based SSDs.

C. Cache Replacement Policies for Flash-based SSDs

In recent years, many cache replacement algorithms have been proposed to improve the efficiency of ca che management for SSDs. We will present two cache replacement algorithms for SSDs, which are most relevant to this study.

Based on the asymmetric read and write costs in delay and energy consumption of flash-based SSDs, a Clean First LRU (CFLRU) cache replacement policy was proposed in [19]. CFLRU divides the LRU list into a working region and a clean-first region. It evicts clean pages preferentially in the clean-first region until the number of pages that hit in the working region reaches a suitable level. In other words, CFLRU reduces the number of costly write and potential erase operations by trading off the number of reads as long as the degradation of cache hit ratio does not degrade the performance.

As revealed by previous studies, there are serious contention for the flash resources and the severe mutually adversary interference between the user I/O requests and GC-induced I/O requests in SSDs. In order to address this important performance issue in flash-based storage systems, a Garbage Collection aware Replacement policy (GCaR) was proposed in [24]. Different from the traditional cache management schemes, GCaR not only exploits the locality of workloads, but also takes the *Miss_Penalty* as an important design factor in the cache replacement. The basic idea is to give higher priority to cache the data blocks belonging to the flash chips that are currently doing GC. This substantially lessens the contentions between the user I/O operations and the GC-induced I/O operations.

D. Motivation

For the on-board cache management within the flash-based SSDs, the access times depend on the workload on flash chips. However, existing cache replacement algorithms, such as CFLRU, are not aware of the variance of *Miss_Penalty*. They assume that the access times to low-level flash chips are consistent. The GCaR algorithm considers the GC state of flash chips, which is orthogonal and can be easily incorporated into existing cache management algorithms. But it becomes effective only when GC operations occur. If the flash chips, where the cached data blocks are resident, are all not in the GC state, the GCaR degrades to the normal cache replacement policies. None of them takes the length of I/O queue within flash chips into consideration.

As revealed by aforementioned discussion, the *Miss_Penalty* is very related to the length of queues on chips and it has a significant impact on the system performance, which is also validate in our experiments, shown in Fig. 5 in Section IV. It motivates us to propose a chip-level load-aware cache replacement algorithm to improve the efficiency of equipped cache within flash-based SSDs. Its basic idea is to give a higher priority to cache the blocks on the flash chips that are in overloaded state. When replacing a data block in the cache line, we will check whether the data block belonging to the flash chip that is in overloaded state or not. If so, the data block will be kept in the cache for a longer time until the chip becomes light loaded. Otherwise, it will be replaced as usual. We present the details of our algorithm in Section III.

III. Load-aware Cache Replacement Algorithm

In this section, we first present the main idea of LCR, then describe its implementation details, and finally show its read and write flows.

A. The Main Idea of LCR

Different from traditional cache management policies for flash-based SSDs, LCR not only exploits the locality of workloads, but also takes the *Miss_Penalty* into consideration. From Fig. 2, we can find that the *Miss_Penalty* to an overloaded flash chip may be orders of magnitude larger than flash chips with light load. We should assign a higher priority to the data

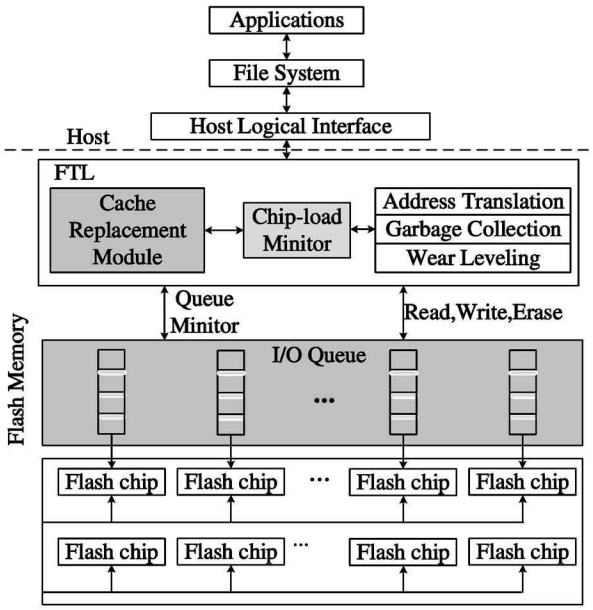


Fig. 3. Overall architecture of LCR scheme.

blocks on a overloaded chip to keep them in the cache. The main idea of LCR is as follows.

- When there is no enough free space for the I/O requests in the cache, we will select the victim data blocks on the chips whose load is minimum.
- We trade off the *Miss_Rate* and the *Miss_Penalty* by adjusting the range of victim blocks. We select victim blocks in the destaging region only, but not the whole LRU list. So it not only maintains the similar cache hit ratio with LRU, but also reduces the computation cost of measuring load on flash chips.
- The priority of a data block in the cache also changes dynamically according to the load level on the chip where it residents. In order to avoid metadata update overhead, we check the load level on a flash chip only when a data block needs to be evicted.

We also implement our scheme LCR to evaluate its performance, the system structure is depicted in Fig. 3. The upper-level host system consists of three components, applications, file system, and host logical interface (HIL). The HIL receives I/O requests from host system and delivers them into the flash translation layer (FTL). The on-board cache management will check whether the requested data is in the cache. They will be serviced by the on-board cache when the data is in the cache. Otherwise, they will be handed by FTL to the low-level flash chips. LCR is composed of two key modules, cache replacement module and load monitor module. The cache replacement module will interact with the load monitor module to determine which data block should be replaced or flushed so as to reduce the *Miss_Penalty*. The load monitor module plays an important role in assessing the real-time load levels on flash chips to decide which are overloaded or light loaded. In the following subsections, we will illustrate the implementation details of LCR.

B. Implementation Details of LCR

1) **Definition of load level L_{chip}:** As discussed before, the load level is decided by three basic operations, read, write and erase. In order to quantify the load levels of flash chips, we formulate the queue delay of user I/O requests and GC-induced I/Os as the load level of flash chips, named L_{chip}.

$$L_{chip} = N_R * T_{read} + N_W * T_{write} + P * T_{gc}, \qquad (2)$$

where N_R and N_W denote the number of read and write requests in a queue, respectively, T_{read} and T_{write} denote the access latencies of processing a read request and a write request, respectively. The value of P is either 1 or 0, which indicates whether the flash chip is in GC state or not. T_{gc} denotes the overhead of GC operation, which includes the time of moving valid pages and erasing the victim block. Realistically $P*T_{gc}$ should be the remaining time to complete GC. However, it is difficult to be captured. Therefore we simply use the definition of T_{gc} in the following experiments.

2) **Data structures for LCR:** To realize the cache replacement scheme, several data structures along with proper algorithms are proposed. To minimize the time to choose the victim data block in cache line, we use a data structure as shown in Fig. 4. The data block nodes are stored in a LRU list, and LCR divides the LRU list into two regions, working region and destaging region. The working region consists of the recently used data blocks and most of the cache hits happen in this region. The destaging region consists of data blocks which are candidates for eviction. The size of the destaging region is called *window size*. LCR preferentially selects a clean data block in the destaging region and evicts it to save flash write cost. If there are no clean data blocks in the destaging region, LCR evicts a dirty block which updated copy is stored in a chip with minimal L_{chip} and flush it to the flash memory.

Fig. 4 shows an example of the LCR scheme. All data blocks are not clean in the destaging region, with LRU scheme, the data block D_4 at the end of the LRU list should be evicted from the cache. However, by checking the mapping information, data block D_4 belongs to flash Chip 2 with $L_{chip} = 20ms$, which are larger than all other ones. So LCR keeps it longer in the cache, and instead evicts data block D_6 on the flash chip with minimal L_{chip} to free cache space.

3) **Search operation:** An I/O request from upper-level file systems mainly consists of a sector number, data size and the type of the operation. The logical page number can be identified by dividing the sector number by the page size, then LCR searches the corresponding data block node in the LRU list. Although it is implemented with a sequential search, the actual cost is not so high since the number of data block nodes in LRU is small due to the limited size of cache. If the I/O request is found in the LRU list, which is served in the cache, the LCR scheme rearranges the LRU list by moving the data block node to the head of the LRU list. Otherwise, LCR immediately terminates the search process.

4) **Insert operation:** When the I/O request is not found in the cache, a new data block node will be allocated and attached

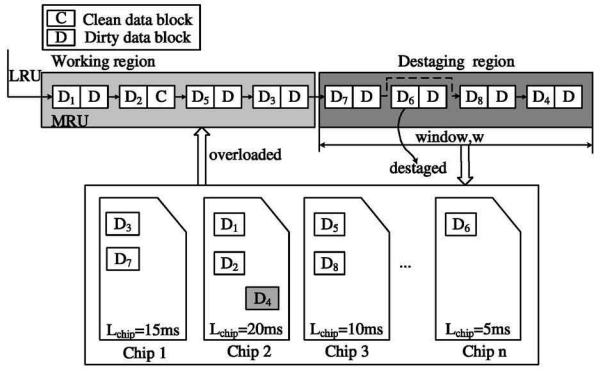


Fig. 4. Data structures of LCR.

to the head of the LRU list. The requested data is retrieved from the flash chips and stored in this node when the request is a read. If it is from a write request, the data is directly written into this node without updating the corresponding flash chip immediately. Although the old data in the flash chip is an invalid version, the subsequent requests to the same data will be serviced in the cache. So the invalid version of the data in the flash chip does not induce to any read error.

5) **Replace operation:** When the cache is full and there is no free space for accommodating a new I/O request, the cache should evict a data block. If the evicted data block is clean, it will be directly discarded. Otherwise, the evicted data block is dirty, means that the data block in the cache is different from the copy in the flash chip. Thus, the victim data block will be destaged into the low-level flash chips. LCR selects a victim block by the following rules in order.

- All the data blocks that are clean in destaging region are taken as candidates and the last one will be selected as a victim block according to the LRU algorithm. This victim bock with clean state is an optimal selection since it will not cause any flash write cost.
- Considering the latency of destaging the data block into the flash chips, the data block on the flash chips with minimal L_{chip} will be selected when there is no clean data block in the destaging region.

In order to reduce the computational cost of L_{chip} for flash chips, LCR searches victim blocks within the destaging region only. The process of selecting a victim block is also presented in Alg. 1.

C. The Read and Write Flow with LCR

Cache plays an important role in smoothing the gap between the upper-level applications and lower-level flash chips. Traditional cache replacement schemes are usually aimed for reducing the miss ratio of the cache by exploiting workload locality. They assume that the *Miss_Penalty* is a constant. However, for flash-based SSDs with different levels of load, the *Miss_Penalty* of accessing the missed data on flash chips with heavy load is evidently larger than that of accessing flash chips with light load. Thus, the cache performance should not be simply evaluated by the hit ratio or miss ratio.

Algorithm 1 Select Victim Block

Input:
1: LRU_List
Output:
2: $VictimBlock$
3: **procedure** SELECT_VICTIM_BLOCK(LRU_List)
4: $x \leftarrow 1$
5: $MinLchip \leftarrow \infty$
6: $CurrentBlock \leftarrow$ GetTailNode(LRU_List)
7: **while** $x <= WindowSize$ **do**
8: $x \leftarrow x + 1$
9: **if** $CurrentBlock$ *is Clean* **then**
10: return $CurrentBlock$
11: **else if** $CurrentBlock.Lchip < MinLchip$ **then**
12: $MinLchip \leftarrow CurrentBlock.Lchip$
13: $VictimBlock \leftarrow CurrentBlock$
14: **end if**
15: $CurretBlock \leftarrow CurrentBlock.Pre$
16: **end while**
17: return $VictimBlock$
18: **end procedure**

Therefore, in the view of flash-based SSDs, both *Miss_Rate* and *Miss_Penalty* are important factors in the LCR scheme. Moreover, the LCR cache replacement policy works not only for write requests, but also for read requests.

If LCR receives a write request from file systems, the LCR scheme will check whether it hits the cache. The data is overwritten in the cache when the request can be found. Otherwise, the LCR scheme will check whether there is free space in cache. If cache is full, LCR will evict data blocks in the cache to make free space for the newly written data. The data blocks in the destaging region will be checked, if there is a clean data block, it will be evicted from the cache and directly discarded; otherwise, the LCR scheme will select the victim data block on the flash chips with the minimal L_{chip} and flush it to the flash chip. The newly written data will be kept in the cache and the LRU list is updated. If the cache has enough free space, a new slot is allocated for the new data, meanwhile, the LRU list is updated. A detailed description of the write flow of LCR is shown in Alg. 2.

For a read request, LCR works similarly. If cache hit happens, LCR immediately returns the cached data to the host. If LCR cannot find the requested data in the cache, it will be fetched from a flash chip and the free cache space availability will be checked. If cache is full, then LCR will evict data blocks in the cache line to make free space for the newly fetched data. The data blocks in the destaging region will be checked, and if there is a clean data block, it will be evicted from the cache and discarded directly, otherwise, the LCR scheme will select a victim data block on the flash chip with minimal L_{chip} and flush it to the flash chips. The newly fetched data will be kept in the cache and the LRU list is updated. If the cache has enough free space, a new slot is allocated for the newly fetched data block, meanwhile, the LRU list is updated. After the requested data block is fetched from the flash to cache, it will be returned to host. A detailed description of the read flow of LCR is shown in Alg. 3.

Algorithm 2 Write flow in LCR

Input: I/O Write Requests: $W_1, W_2, \cdots, W_i, \cdots$

procedure LCR_WRITE(W_i)
  if W_i *is in the cache* **then**
    RemoveHead(LRU_List,W_i)
    Write_to_Cache(W_i)
  else if *cache is full* **then**
    $Victim \leftarrow$ Select_Victim_Block(LRU_List)
    if $Victim.Flag$ *is Dirty* **then**
      Flush_Block_to_Flash($Victim$)
    end if
    Delete($LRU\_List, Victim$)
    Write_to_Cache(W_i)
    Update(LRU_List, W_i)
  else
    Write_to_Cache(W_i)
    Update(LRU_List, W_i)
  end if
end procedure

Algorithm 3 Read flow in LCR

Input: I/O Read Requests: $R_1, R_2, \cdots, R_i, \cdots$

procedure LCR_READ(R_i)
  if R_i *is in the cache* **then**
    RemoveHead(LRU_List,R_i)
    Return_from_Cache(R_i)
  else if *cache is full* **then**
    $Victim \leftarrow$ Select_Victim_Block(LRU_List)
    if $Victim.Flag$ *is Dirty* **then**
      Flush_Block_to_Flash($Victim$)
    end if
    Delete($LRU\_List, Victim$)
    Return_from_Flash(R_i)
    Update(LRU_List, R_i)
  else
    Return_from_Flash(R_i)
    Update(LRU_List, R_i)
  end if
end procedure

IV. Performance Evaluation

In this section, we first introduce the simulation setups and methodology, then present the evaluation results, and finally discuss the impact of different parameter settings on the performance of our scheme LCR.

TABLE II
DiskSim Configuration

Parameter	Value
Page Size	4KB
# of pages per block	64
# of blocks per plane	512
# of planes per package	8
Page read latency	0.025ms
Page write latency	0.200ms
Block erase latency	1.500ms

TABLE III
Statistics of I/O workloads

Trace	Total # of requests	Write Ratio	Avg. Size
web_0	2029945	70.12%	7.64 KB
ts_0	1801734	82.42%	4.64 KB
proj_0	4224524	87.52%	19.06 KB
wdev_2	181266	99.89%	4.08 KB
financial2	3698866	17.65%	2.42 KB

A. System Configuration

We employ DiskSim [3] with SSD extension [2] to evaluate our scheme. This SSD simulator has been widely applied in the exploration of SSD-based storage systems. In this paper, we implemented our proposed LCR scheme by integrating it into the open-source SSD simulator. We modeled an SSD of 64GB, which is configured with 64 chips. Greedy garbage collection and dynamic wear leveling are implemented, and the over-provisioning ratio is set as 15% of the SSD's capacity, which are default settings for most of SSDs. Other main parameters of the SSD are listed in Table II.

We use realistic enterprise scale workloads to study the performance impact of different cache replacement policies. The workloads are chosen from MSR Cambridge traces on servers at Microsoft Research Cambridge Lab [1], which are also widely used in previous works to evaluate the performance of flash-based SSDs [5]. The main workload parameters of the four traces are summarized in Table III.

In the evaluations, we compare LCR with three algorithms, LRU, CFLRU and GCaR-CFLRU. Note that LRU is most common cache replacement algorithm and GCaR-CFLRU can be regarded as the state-of-the-art algorithm which further improves CFLRU by taking GC interference into consideration. We use the average response time as the user performance metric to evaluate the effectiveness of LCR against other three schemes. We take the LRU scheme as the baseline. The key characteristics of the CFLRU and GCaR-CFLRU are summarized below.

- **CFLRU** [19] is a cache management scheme for flash-based SSDs. Based on the asymmetric performance of read and write operations of flash memory. It prefers to choose a clean data block rather than a dirty one as a victim since the delay of writing a dirty data block is significantly longer.
- **GCaR-CFLRU** [24] aims to substantially reduce the contention and interference between the user I/O operations and the GC-induced I/O operations. The basic idea is to give a higher priority to cache the blocks on the flash chips that are in GC state.

B. Performance Results

The response time to a I/O request mainly consists of three parts, delivering time on data bus (T_b), processing time on chip (T_c), and latency of waiting in chip queue (T_w). Note that T_b and T_c are linearly related to the size of a request, whose transfer rates are determined by hardware property. We believe that T_w is the only part that can be improved. In our

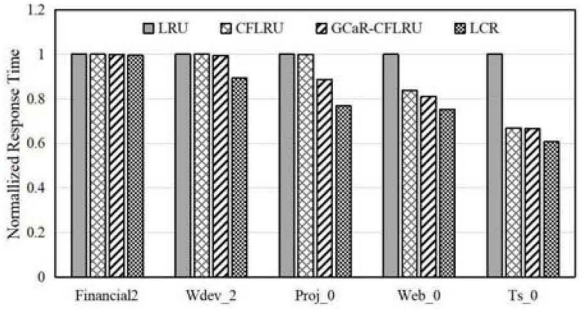


Fig. 5. Average response time of LRU, CFLRU, GCaR-CFLRU and LCR

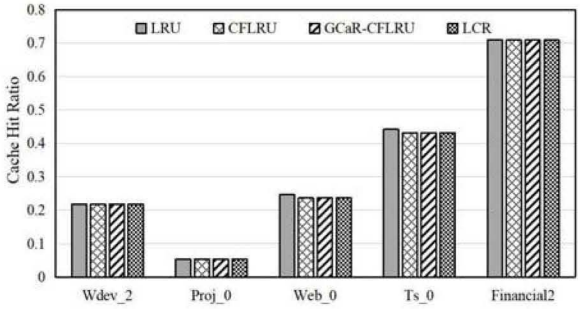


Fig. 7. The cache hit ratio with LRU, CFLRU, GCaR-CFLRU and LCR

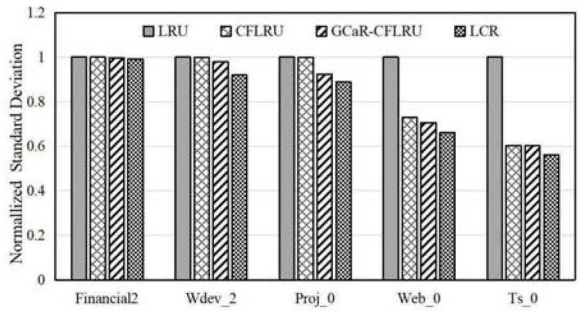


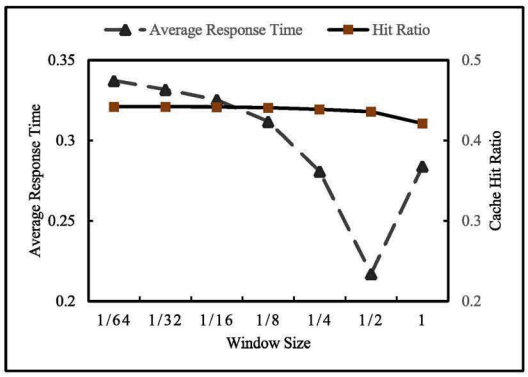
Fig. 6. Standard deviation of response times with LRU, CFLRU, GCaR-CFLRU and LCR

proposed cache replacement scheme LCR, we prefer to choose a data block on the flash chips with light load to reduce the factor T_w of the destaging victim data blocks. To evaluate the performance of various algorithms, we first record the response time to each I/O request in simulations, then show the average response time and their standard deviation.

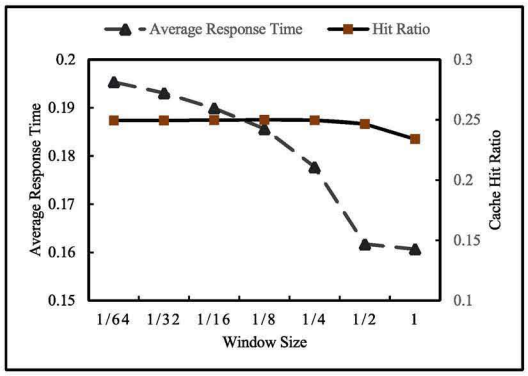
1) **The average response time of I/O requests.** Fig. 5 shows the average response time. We normalize the result of LRU as one. We can see that LCR outperforms all existing schemes, including LRU, CFLRU, and GC-CFLRU. In particular, for the proj_0 and wdev_2 traces, LCR reduces the average response time by up to 23.7% (12.3%) and 11.6% (10.1%) compared to LRU (GCaR-CFLRU), respectively. We also measure the ratio of evicted dirty data blocks for the proj_0 and wdev_2 traces, and find that the ratio is significantly high, nearly more than 95%. Thus, CFLRU does not work well in this case because there are few clean data blocks in the clean-first region. And the data volume of the wdev_2 trace is small, so there are few GC operations. As a result, GCaR-CFLRU shows inefficiency in this trace, but LCR still shows better performance. For proj_0 trace with more frequent GC operations, GCaR-CFLRU shows its advantage in reducing the average response time by up to 11.4% compared to the baseline LRU scheme, LCR performs even better than GCaR-CFLRU, and it reduces the average response time by up to 23.7% compared to LRU. As discussed in Section II, the main reason is that apart from GC operations, the unbalanced loads on different flash chips also play an important role in the response time to user I/O requests. On the other hand, for web_0 and ts_0 traces, there are 76% and 40% replacing ratio of the clean data blocks. Thus, CFLRU scheme already performs well, and GCaR-CFLRU does not improve the performance over CFLRU obviously, the only reason is that the two traces induce only a very small portion of GC operations. However, LCR still shows an obvious improvement, with a reduction of average response time compared to LRU and GCaR-CFLRU by up to 24.8%(5.2%) and 39.2%(5.9%), respectively. But for financial2 trace, CFLRU, GC-CFLRU, and LCR have almost no improvements compared to LRU. The important reason is that financial trace is mainly read operations, inducing a few dirty data blocks and a very small portion of GC operations. It is clear that, by avoiding evicting the data blocks on the chips with heavy load, LCR scheme are able to significantly reduce user response time. The results indicate that unbalanced loads on flash chips have a significant impact on user response time.

2) **The standard deviation of response time.** Fig. 6 shows the normalized standard deviation of response times with LRU, CFLRU, GCaR-CFLRU and LCR. The standard deviation of response times is used to measure the degree of dispersion on the response time of I/O requests. The smaller the value, the better system performance. From Fig. 6, we can see that LCR also performs the best under all workloads, but for proj_0 trace, the reduction of the standard deviation of response time is not too large, e.g., it is 11.2% (5.1%) compared with LRU (GCaR-CFLRU), and similar results can be observed for wdev_2 trace. Meanwhile, for ts_0 trace, LCR reduces the standard deviation of response times by up to 43.9% (5.4%) over LRU (GCaR-CFLRU), and the web_0 trace shows the same improvement of the system performance. This is because we take into consideration the unbalanced loads on flash chips when evicting data blocks. It plays an important role in smoothing the response time of I/O requests and reducing the access latency. As a result, LCR can substantially alleviate the problem of long tail on access latency.

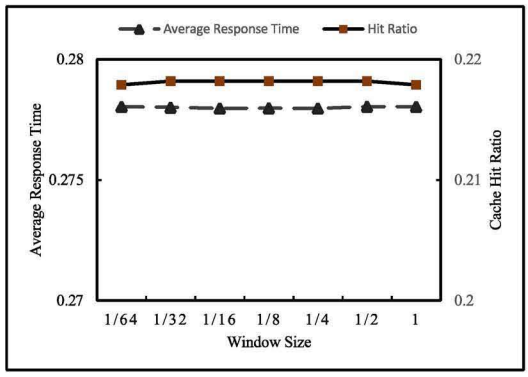
3) **The cache hit ratio.** We now evaluate the cache hit ratio of different schemes driven by the four realistic traces. Fig. 7 shows the cache hit ratios for LRU, CFLRU, GCaR-CFLRU and LCR. It is clear that all of them have a similar performance in cache hit ratio. This is because the main objectives of CFLRU, GCaR-CFLRU and LCR are not to



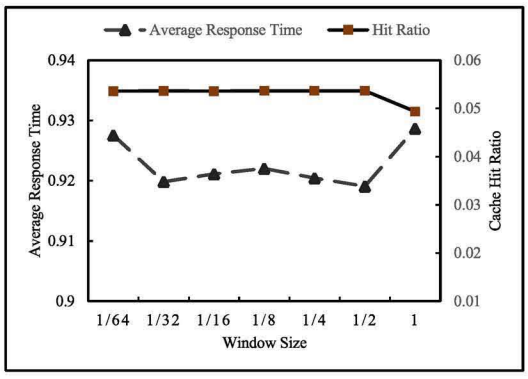
(a) TS_0 trace



(b) WEB_0 trace



(c) WDEV_2 trace



(d) PROJ_0 trace

Fig. 8. The average response time and cache hit ratio of LCR with various window size for realistic traces.

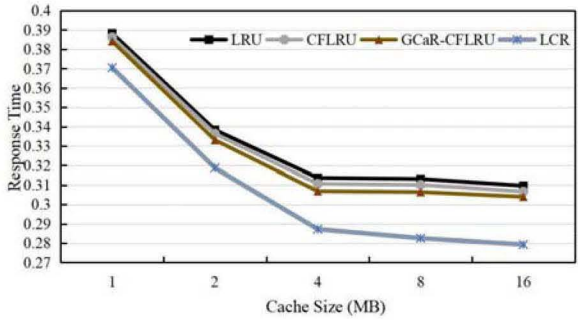


Fig. 9. Average response time varying the cache size

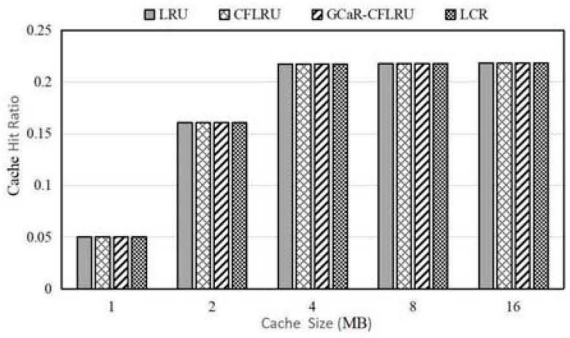


Fig. 10. Cache hit ratio varying the cache size

improve the cache hit ratio. On the other hand, compared with the result of the average response time in Fig. 5, we can see that even with a similar cache hit ratio, the average response time may vary significantly. The main reason is obvious as the miss penalty may vary when evicting data blocks belonging to different flash chips. Actually, the miss penalty on flash chips with heavy load is much higher than that on light ones. This phenomenon further implies that cache hit ratio is not the only factor which imposes on user I/O request response time, and it is also the reason why improving cache hit ratio should not be the only design objective of an effective cache scheme for flash-based SSDs.

C. The effects of cache Size

We have analyzed the impact of cache size on average response time and cache hit ratio for all of the five traces. The results of all traces show the similar trends. Due to the limited space, we only show the experimental results of wdev_2 trace. Fig. 9 shows the average response time for each evaluated schemes by varying the size of the cache from 1MB to 16MB. In general, the average response time decreases as the cache size increases. When the cache size is larger than 4MB, the average response times become stable. Overall, LCR shows better performance than the rest of the cache management schemes for all cache sizes. We can see that LCR reduces the average response times by 11.6%, 10.5%, 9.8% on average compared with the LRU, CFLRU, and GC-CFLRU schemes

respectively. It is clear that, by avoiding evicting the cached data blocks to the overloaded flash chips, the LCR is able to significantly reduce user response times. The results indicates that *Miss_Penalty* can not be ignored.

Fig. 10 shows the cache hit ratio for each cache replacement algorithms with cache size varying in a large range. As seen, greater cache size improves the cache hit ratio, when the cache size is larger than 4MB, the cache hit ratio becomes stable. Moreover, we can see that LRU, CFLRU, GC-CFLRU and LCR have a similar performance in cache hit ratio. It is obvious that except for LRU, the other schemes try to reduce the miss penalty. As a result, the cache hit ratio does not get any improvements. On the other hand, the cache hit ratios are not consistent with the average response times, as shown in Fig. 9. This phenomenon further implies that cache hit ratio is not the only factor related to the overall performance.

D. Impact of Window Size

To find the impact of window size on the performance of LCR, we further show the results by varying the window size of the destaging region from 1/64, 1/32, ..., to 100% of the total cache size.

1) **Impact on average response time.** The results are shown in Fig. 8, and we can find that for ts_0 and web_0 traces, window size shows an evident impact on the performance of LCR. When the window size varies from 1/64 to 1/2, that is, when the window size becomes larger, the average response time becomes smaller. On the one hand, with the window size being increased, more clean data blocks can be found in the destaging region. On the other hand, the probability of choosing a victim data block on flash chips with light load increases. But the computational cost of L_{chip} may increase dramatically if the window size grows too large. For the wdev_2 and proj_0 traces, the performance of LCR is stable for different window sizes. So different workloads have different sensitivity to the window size, which depends on the essential features of workloads. For each workload, there should be an optimal setting of window size which minimizes the average response time. Therefore, ideally, this window size should be tuned dynamically to adapt to different workloads and cache size. In the absence of such a dynamic and adaptive mechanism to tune the window size (which is our future work), we set the window size to a fixed value in our current evaluation study.

2) **Impact on cache hit ratio.** Fig. 8 further shows the impact of window size on cache hit ratio. We can see that the cache hit ratio of LCR is not sensitive to the window size for all workloads. The main reason is that the first objective of LCR is not to improve the cache hit ratio. Note that the design of LCR scheme is based on LRU, but we made a trade off between the *Miss_Rate* and *Miss_Penalty*, and this not only achieves a reduction in average response time, but also maintains the cache hit ratio similar to LRU. As a result, the window size has no big impact on the cache hit ratio of LCR.

V. Conclusions

As revealed by previous studies, the loads on different flash chips, may be seriously unbalance. The load on a chip comes from the length of I/O requests queue and GC overhead. Thus, the *Miss_Penalty* to a flash chip with heavy load is longer than that to one with light load. In this paper, we propose a Load-aware Cache Replacement algorithm, called LCR, to improve the performance of flash-based SSDs. The basic idea is to give a higher priority to cache the blocks on the flash chips with heavy load. This substantially improve the system performance. We have implemented our scheme with the SSD extended Disksim simulator and conducted extensive evaluations. The performance results show that LCR can evidently reduce the average response time over the existing cache replacement algorithms.

Acknowledgment

This work was supported in part by the National Key R&D Program of China 2018YFB1003204. The work of Yubiao Pan was supported in part by the Fujian Province Natural Science Foundation under Grant No. 2018J05107 and the Education Department Foundation of Fujian Province under Grant No. JAT170039.

References

[1] SNIA. IOTTA reposity. http://www.snia.org/.

[2] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy. Design tradeoffs for ssd performance. In *Proceedings of USENIX Annual Technical Conference*, volume 57–70. USENIX Association, 2008.

[3] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. The disksim simulation environment version 4.0 reference manual. *Parallel Data Laboratory*, page 26, 2008.

[4] X. Ding, S. Jiang, and F. Chen. A buffer cache management scheme exploiting both temporal and spatial localities. *ACM Transactions on Storage (TOS)*, 3(2):5:1–5:27, 2007.

[5] C. Gao, L. Shi, M. Zhao, C. J. Xue, K. Wu, and E. H.-M. Sha. Exploiting parallelism in i/o scheduling for access conflict minimization in flash-based solid state drives. In *In Mass Storage Systems and Technologies (MSST), 2014 IEEE 30th Symposium on*, pages 1–11. IEEE, 2014.

[6] A. Gupta, Y. Kim, and B. Urgaonkar. DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, volume 44, pages 229–240. ACM, 2009.

[7] J. Hu, H. Jiang, and P. Manden. Understanding performance anomalies of ssds and their impact in enterprise application environment. In *Proceedings of ACM SIGMETRICS Performance Evaluation Review*, pages 415–416. ACM, 2012.

[8] J. Hu, H. Jiang, L. Tian, and L. Xu. PUD-LRU: An erase-efficient write buffer management algorithm for flash memory ssd. In *Proceedings of 2010 IEEE International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems*, pages 69–78. IEEE, 2010.

[9] J. Hu, H. Jiang, L. Tian, and L. Xu. GC-ARM: Garbage collection-aware ram management for flash based solid state drives. In *Proceedings of 2012 IEEE 7th International Conference on Networking, Architecture and Storage*, pages 134–143. IEEE, 2012.

[10] A. Jain and C. Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *Proceedings of 2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture*, pages 78–89. IEEE, 2016.

[11] 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*, pages 8–8. USENIX, 2005.

[12] S. Jiang and X. Zhang. LIRS: an efficient low interreference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[13] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee. FAB: flash-aware buffer management policy for portable media players. *IEEE Transactions on Consumer Electronics*, 52(2):485–493, 2006.

[14] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th VLDB Conference*, pages 439–450. Morgan Kaufmann, 1994.

[15] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha. LRU-WSR: integration of lru and writes sequence reordering for flash memory. *IEEE Transactions on Consumer Electronics*, 54(3):1215–1223, 2008.

[16] M. Jung and M. T. Kandemir. Sprinkler: Maximizing resource utilization in many-chip solid state disks. In *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, pages 524–535. IEEE, 2014.

[17] M. Jung, E. H. Wilson III, and M. Kandemir. Physically addressed queueing (paq): improving parallelism in solid state disks. In *ACM SIGARCH Computer Architecture News*, volume 40, pages 404–415. IEEE Computer Society, 2012.

[18] N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 3rd conference on USENIX Conference on File and Storage Technologies*, pages 115–130. USENIX, 2003.

[19] S.-y. Park, D. Jung, J.-u. Kang, J.-s. 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*, pages 234–241. ACM, 2006.

[20] A. Selvan Ramasamy and P. Karantharaj. RFLRU: A buffer cache management algorithm for solid state drive to improve the write performance on mixed workload. *Engineering Letters*, 22(4):158–169, 2014.

[21] Z. Shen, J. Shu, and P. P. Lee. Reconsidering single failure recovery in clustered file systems. In *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*, pages 323–334. IEEE, 2016.

[22] H. Shim, B.-K. Seo, J.-S. Kim, and S. Maeng. An adaptive partitioning scheme for dram-based cache in solid state drives. In *Proceedings of 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies*, pages 1–12. IEEE, 2010.

[23] Q. Wei, C. Chen, and J. Yang. CBM: A cooperative buffer management for ssd. In *Mass Storage Systems and Technologies (MSST), 2014 30th Symposium on*, pages 1–12. IEEE, 2014.

[24] 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*, pages 28:1–28:12. ACM, 2016.

[25] M.-C. Yang, Y.-M. Chang, C.-W. Tsao, P.-C. Huang, Y.-H. Chang, and T.-W. Kuo. Garbage collection and wear leveling for flash memory: Past and future. In *Proceedings of 2014 International Conference on Smart Computing*, pages 66–73. IEEE, 2014.

[26] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of USENIX Annual Technical Conference*, pages 91–104. USENIX Association, 2001.