DRAM Cache Management with Request Granularity for NAND-based SSDs

Haodong Lin
linhaodong@email.swu.edu.cn
Southwest University of China

Zhibing Sha
shazb171318515@163.com
Southwest University of China

Jun Li
lijun19991111@126.com
Southwest University of China

Zhigang Cai
czg@swu.edu.cn
Southwest University of China

Balazs Gerofi
balazs.gerofi@intel.com
Intel Corporation, USA

Yuanquan Shi
syuanquan@163.com
Huaihua University of China

Jianwei Liao*
liaotoad@gmail.com
Southwest University of China

Abstract

Most flash-based solid-state drives (SSDs) employ an on-board Dynamic Random Access Memory (DRAM) to cache hot data at the SSD page granularity. This can significantly reduce the number of flush operations to the underlying arrays of SSDs given that there is sufficient locality in the applications' I/O access pattern. We observe, however, that in most I/O workloads over SSDs the buffered data of small sized requests are more likely to be re-accessed than those of larger requests, which also require more DRAM space for caching their data.

To improve the efficiency of the DRAM cache inside SSDs, this paper presents a request granularity-based cache management scheme, called ***Req-block***. The proposed mechanism manages cached data according to the size of write requests and supports multi-level linked lists for sifting the cached data blocks (termed as request blocks), by taking both their size and hotness into account. Comprehensive evaluation shows that our proposal improves cache hits by up to 144.0%, and decreases I/O latency by 14.3% on average, compared to existing state-of-the-art SSD cache management schemes.

CCS Concepts: • **Computer systems organization → Secondary storage organization**.

*Corresponding author.

ICPP '22, August 29-September 1, 2022, Bordeaux, France

ACM ISBN 978-1-4503-9733-9/22/08…$15.00
https://doi.org/10.1145/3545008.3545081

Keywords: Cache management, NAND-flash, solid-state disks

ACM Reference Format:
Haodong Lin, Zhibing Sha, Jun Li, Zhigang Cai, Balazs Gerofi, Yuanquan Shi, and Jianwei Liao. 2022. DRAM Cache Management with Request Granularity for NAND-based SSDs. In *51st International Conference on Parallel Processing (ICPP '22), August 29-September 1, 2022, Bordeaux, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3545008.3545081

1 Introduction

NAND-based solid-state drivers (SSDs) show better performance on random access, power consumption, and massive parallelism compared with hard drives (HDDs) [1, 2]. Especially, with the drop of per-GB cost, SSDs are gradually replacing HDDs and have been widely used in digital devices, data centers, and high-performance computers [3, 4].

Due to their compact structure, however, the limit of program/erase cycles (P/Es) of high-density SSDs decreases greatly (e.g. SSDs with quadruple level cell support 500 P/Es), and thus impacts the endurance of SSDs [5] [6]. For achieving better overall performance, a faster but smaller capacity Dynamic Random Access Memory (DRAM) is equipped inside SSD devices [7]. Consequently, we can avoid flushing write data to the underlying flash cells by absorbing them in the DRAM cache, which in turn enhances the lifetime of the device [8].

Figure 1 shows the typical architecture of modern NAND-based solid-state drives [10]. The DRAM cache is generally deployed between the host interface logic (*HIL*) and the flash array. Besides storing the mapping table to support address translation, the DRAM cache is commonly used to buffer (hot) write data of user applications [12]. As seen in the figure, when a write request is delivered to the SSD from the *HIL* interface, the flash translation layer (*FTL*) services the request by buffering the data in the cache so that the latency of the write operation can be significantly shortened [11, 14].

Once the DRAM cache becomes full with buffered data, the cache management schemes need to evict in-cache data

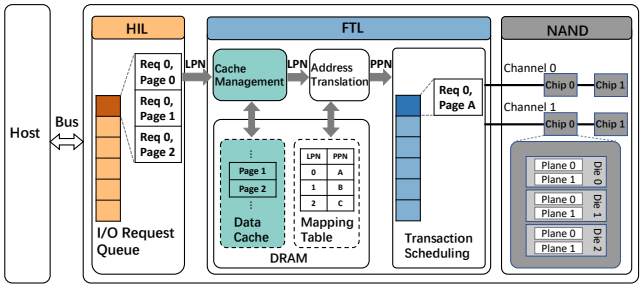

Figure 1. Architectural overview of a modern NAND-based solid-state drive [10]. *A read/write request (not sync) is being served while its data is obtained/buffered from/into the data cache.*

pages and flush them to flash array for making room for newly written data. Most traditional cache management approaches inside SSDs, such as *LRU*, *CFLRU* and their variations [9], manage the cached data at the page granularity that corresponds to the basic read/write unit of the flash array.

In order to better enhance cache use efficiency, certain advanced cache management schemes have been proposed to match the features of modern flash memory [11]. Considering that the garbage collection (GC) process of flash memory is performed at the unit of a block that consists of multiple SSD pages, SSD block-level cache management approaches including *FAB* [19], *BPLRU* [15], and *PUD-LRU* [21] have been proposed to better exploit spatial locality.

The SSD page-based or SSD block-based cache management schemes evict the buffered data by referring to the degree of temporal or spatial locality, while the features of I/O requests in workloads are neglected [11]. After analyzing a large number of I/O workloads of real-world applications, we observed that the cached data pages associated with small sized I/O requests are more likely to be accessed again, in contrast to those associated with larger size requests.

Therefore, we propose a novel cache management algorithm (called *Req-block*) that deals with the cached data at the granularity of write requests, and each cached item is named as a request block (having one or more data pages). Specifically, our scheme supports multi-level linked lists for sifting the cached request blocks, by taking both the size and the hotness of write requests into consideration. This boosts cache hits and improves I/O performance of SSDs. In summary, this paper makes the following contributions:

- We introduce a request-granularity based cache management scheme for SSD devices. Considering that small sized requests' data are more likely to be accessed compared to those with larger sizes in the DRAM cache, our approach manages the cached data pages at the granularity of write requests, which we call request blocks. The proposed scheme makes different cache adjustments for the cached request blocks based on the size of the write requests. In particular, it preferentially keeps small request blocks in the DRAM cache when making space for newly written data.
- We maintain three-level linked lists for sifting the cached request blocks, to support cache replacement. Specifically, newly written data will be organized into a request block and inserted into the lowest level list. A request block will be upgraded to higher level linked list if it has been hit, or it will be split first and only the hit part of the request block is upgraded. In the eviction process, the scheme selects a victim from the tail request blocks of three linked lists by comparing their request size and access history.
- We evaluate the proposed method by replaying several I/O traces of real-world applications. Experimental results demonstrate that our method improves cache hits by `23.5%` on average and decreases I/O response time by up to `73.9%` in contrast to *LRU* and other state-of-the-art cache management schemes.

The rest of this paper is organized as follows: Section 2 describes the related work on cache management inside SSDs and the motivation of our proposal. In Section 3, we present the design and implementation of the proposed *Req-block* algorithm. The performance of our approach is evaluated and discussed in Section 4. Finally, we make concluding remarks in Section 5.

2 Related Work and Motivation

2.1 Related Work

Caches inside SSDs can absorb certain I/O requests to avoid forwarding them onto the underlying flash array which in turn improves SSD performance [18]. Cache management mainly focuses on the replacement strategy, to make room for newly written data by evicting some of the previously buffered data. Traditional cache management schemes, such as first in first out (*FIFO*), *LRU*, and *LFU* [24], are built on the top of references of temporal locality [22] [23], and can be also naturally applied in SSDs.

Specifically, concerning the scenario of SSDs, advanced cache management schemes can be classified into ***page-based*** and ***block-based*** approaches according to the management granularity.

The *page-based* cache management approaches, such as *CFLRU* [9], *ECR* [10], and *Co-Active* [11], are committed to minimizing the negative effects of cache eviction, considering a page as the basic read/write granularity [17]. According to the asymmetrical eviction cost of clean and dirty pages, *CFLRU* divides the cache space into the working region and the clean-first region. In an eviction process, the least recently used clean page in the clean-first region is preferred to be selected as a victim, since it has the lowest eviction

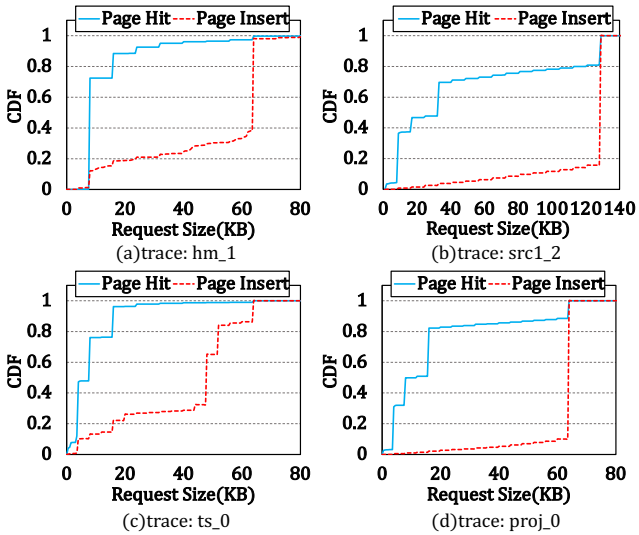

Figure 2. Cumulative Distribution Function (CDF) of page insert and hit statistics as the function of request sizes using a 16MB data cache and the *LRU* replacement policy.

cost. Moreover, *ECR* chooses the victim page which requires the shortest waiting time to be flushed onto the flash cell, by referring to the length of I/O queues of SSD channels (or chips). Similarly, *Co-Active* offers a means to make full use of the internal parallelism of SSDs under I/O write-intensive workloads. Consequently, the cold dirty pages can be actively evicted from the cache and written into flash cells when SSD becomes idle.

To better utilize spatial locality of reference, a number of *block-based* cache management schemes, such as *FAB* [19], and *BPLRU* [15] have been proposed to cluster data pages having spatial correlation, with the granularity of an SSD block. Specifically, *FAB* targets portable media player applications that have intensive sequential write requests. It groups cached data pages in the same flash block and evicts the group that holds the largest number of pages when the buffer is full. However, *FAB* only considers the group size while neglecting data recency [20]. Similarly, *BPLRU* adopts a block-level LRU list and each block can be preferentially evicted from the cache if it is written sequentially, considering that these blocks have the least possibility of being rewritten in the near future. Furthermore, *BPLRU* supports a page padding approach to minimize the merge operations.

More importantly, Du et al. [16] proposed a buffer management scheme called *VBBMS* that takes the different localities of random and sequential requests into account. Specifically, it divides the cache into random and sequential regions for different kinds of requests, and respectively employs *LRU* and *FIFO* for selecting the victims of virtual blocks to be evicted in two regions of the cache. Moreover, it manages the cached data pages at the granularity of virtual blocks for the utilization of spatial locality.

In brief, either *block-based* or *virtual-block-based* cache management approaches work efficiently in their target application contexts, but cannot yield good I/O performance under random access dominated workloads [11]. This is because the small size random requests are unable to make full use of a block granularity with dozens of pages.

2.2 Motivation

In I/O workloads of real-world applications, small write requests commonly reveal higher temporal locality than those with larger write requests [25]. That is to say, the access frequency of a data page may be related to the size of its write request. For quantifying the relationship between the hits of cached data pages and the size of their corresponding write request in the workloads of SSDs, we have conducted a series of experiments. The specification of the experimental platform and the details of the used benchmarks are described in Section 4.1.

Figure 2 shows the results of page hits and page inserts as cumulative distribution function (*CDF*), according to the size of the requests. In the figure, *Request Size* means the size of write request whose contents (i.e. data pages) are inserted into the DRAM cache. The *CDF* result of *Page Hit* implies the proportion of page hits on the pages whose write request size is equal to or less than the corresponding request size, to all hits. Similarly, the *CDF* result of *Page Insert* represents the portion of inserted pages whose request size is equal to or less than the corresponding request size, to all pages inserted to the cache.

In most of the block traces, as shown, the buffered data pages of small size requests[1] contribute about `80%` of the page hits from the entire workload. On the other hand, the data pages from large write requests occupy the majority of the cache space. Taking the traces of *hm_1* and *proj_0* (see Figures 2(a) and 2(d)) as examples, we see that the small size write requests take up less than `20%` of the cache space, but contribute more than `80%` of the page hits.

> ***Observation ❶:*** *Cache hits for requests with small sizes account for a major part of all hits, meanwhile the large size requests take up most of the cache space.*

Furthermore, we exploit how many cached data pages of large size requests are hit in the cache, and Figure 3 presents the results. As shown, only `22.0%-37.2%` of cached data pages of large size requests have been re-accessed.

[1]We refer a small request while its size is not larger than the average size of all requests of selected traces in this analysis, and other requests are regarded as large size requests.

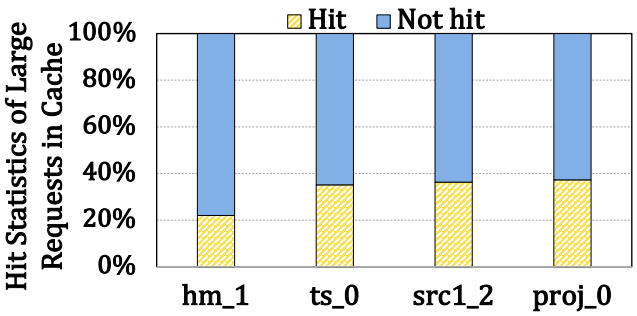

Figure 3. Hit statistics of large requests in cache, with 16MB data cache and the *LRU* replacement policy.

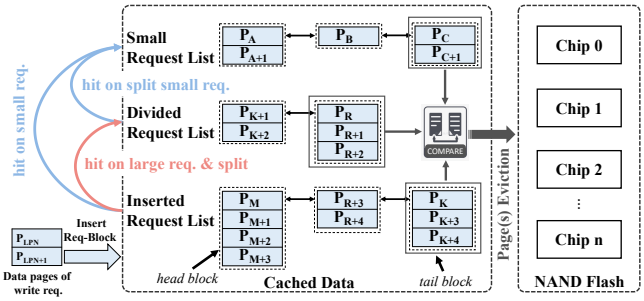

Figure 4. Overall architecture of our proposal of *Req-block* cache management.

> ***Observation ❷:*** *Only a small part of buffered data pages of large size write requests will be re-accessed in the cache.*

Such observations motivate us to design a request size-based cache management approach, to not only assign a high priority for the cached data pages associated with small size requests, but also abstract the hot accessed data from large write requests and keep them separately in the cache.

3 Design and Implementation

3.1 High-Level Overview of Req-block

We propose a DRAM cache management scheme with write request granularity for NAND-based SSDs, called *Req-block*. The principle idea of *Req-block* is taking a request block as the basic unit for cache space allocation, and adopting different techniques for cached request block management with small and large size write requests.

Figure 4 shows a high-level overview of *Req-block*, which holds three-level linked lists for enabling request-granularity based cache management inside SSDs. As seen, when a write request arrives, we build a request block by grouping its data pages and initially insert it to the head of *Inserted Request List (IRL)*. Subsequently, *Req-block* supports upgrading the cached request blocks to higher level linked lists of *Small Request List (SRL)* or *Divided Request List (DRL)*, while they or part of them are hit in the cache.

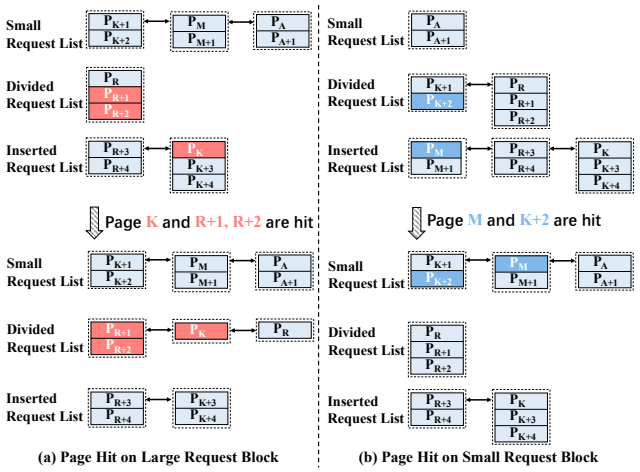

Figure 5. Page hit on small and large request blocks. Assuming the size limit of request block in *Small Request List* (i.e. δ) is set to 2.

Specially, we configure a size limit (i.e. the **parameter** of δ) for the request blocks in *SRL*, and the request blocks that have more than δ data pages cannot be directly moved to *SRL*. We prefer buffering the data pages of small request blocks of *SRL* in the cache, by assigning a higher priority to them.

Once the cache becomes full, *Req-block* compares the priorities of the tail nodes of three linked lists and evicts a request block that has the lowest priority for making available cache space. To this end, the data structure of request block contains the information of the size and the access count, which is used to select the eviction victim.

3.2 Block Request Upgrade

All request blocks of write requests are initially inserted into the *Inserted Request List*. After the block or a part of the block is re-accessed, the proposed *Req-block* scheme will carry out an upgrading adjustment of the request block.

The design principle is that the data pages of small requests have more opportunities to be upgraded to higher level linked lists and thus preferably are kept in the cache. Therefore, *Req-block* supports different routines to adjust small request blocks (their size is not larger than δ) and large request blocks, after they have been re-accessed in the cache.

3.2.1 Hit on Large Request Blocks. A large request block includes a large number of pages, in which some pages are less likely to be re-accessed in the future. *Req-block* splits large request blocks and makes an adjustment for the hit data pages to *Divided Request List*. Note, that the split request block that is originally from a large block request must be inserted to the head of the *DRL* list, regardless of its size and the location of the original request block.

Figure 5(a) shows an adjustment example of page hits on the large request blocks located in either *IRL* or *DRL*. As

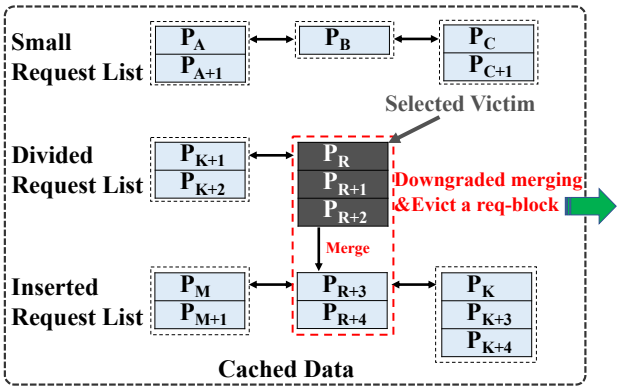

Figure 6. Downgraded merging of cached pages to support eviction with a (large) request block granularity.

seen, the hit data pages will be abstracted from the original request block, and is inserted to the *DRL* list after creating a new request block.

3.2.2 Hit on Small Request Block. When one or more data pages of a small request block (i.e. the page number is not larger than δ) are hit in the cache, *Req-block* adjusts it with an upgrade operation and directly moves the hit small request block to the head of *Small Request List*.

Figure 5(b) illustrates an instance of upgrading two hit small request blocks. As seen, we move the small request block that contains **Page M** to *SRL* from *IRL*, and adjust the split small request block that has **Page K+1** to *SRL* from *DRL*.

3.3 Cache Eviction

Evicted data pages should be selected and then flushed onto underlying flash cells of the SSD to make cache space for new data. In *Req-block*, we introduce data eviction with the unit of request block, in which all data pages belonging to the same request block are evicted in batch to utilize the parallelism of multiple channels inside SSDs.

As previously demonstrated in Figure 4, the victim request block is chosen from the tail nodes of three linked lists, and the request block with the lowest priority will be selected. To this end, we introduce a method to estimate the priority for a given request block, by considering the average access count of each page and the distance between the insert time and current time.

Equation 1 defines the access frequency of request block, labeled as $Freq_{req\_blk}$, and a tail request block with the smallest value of $Freq_{req\_blk}$ will be evicted.

$$Freq_{req\_blk} = \frac{Access_{cnt}}{Page_{num} \times (T_{cur} - T_{insert})} \quad (1)$$

where $Access_{cnt}$ is the access count of the block request since it was buffered in the cache, which is initialized to 1; $Page_{num}$ is the number of pages in the block request; T_{cur} and T_{insert} are the current time point and the insert time of the request block.

Algorithm 1: The Workflows of *Req-block*

Input: I/O request R with request type R_{type}, logical page number R_{lpn}, and request size R_{size}
Output: *NULL*

```
1  Function create_req_blk(list, req)
2      if list.head.Req ≠ req then
3          /*head block of list does not belong to req*/
4          blk = new request block;
5          blk.Req = req;
6          insert_to_head(list, blk);
7  Function get_victim()
8      /*select a victim block*/
9      victim = tail block with minimum Freq_discnt;
10     if victim is a split block then
11         /*merge operation for evicting a request block*/
12         if original block of victim is still in IRL then
13             victim = merge(victim, orig_block);
14     return victim;
15 Function main_routine()
16     lpn ← R_lpn; size ← R_size
17     while size ≠ 0 do
18         /*page hit*/
19         if is_in_cache(lpn) then
20             read or update lpn in cache;
21             request_blk = get_blk(lpn);
22             if request_blk.Page_num ≤ δ then
23                 /*move request block to head of SRL*/
24                 move_to_head(SRL, request_blk);
25             else
26                 remove_from_req_blk(lpn);
27                 create_req_blk(DRL, R);
28                 insert_to_head_blk(DRL, lpn);
29         /*page miss*/
30         else
31             if R_type is write then
32                 if cache_is_full() then
33                     /*eviction for making space*/
34                     victim = get_victim();
35                     evict_from_cache(victim);
36                 create_req_blk(IRL, R);
37                 insert_to_head_blk(IRL, lpn);
38             else
39                 read_from_flash(lpn);
40         size−−; lpn++;
```

Our proposal of *Req-block* adopts three-level linked lists to manage the cached request blocks, and the blocks in *SRL* and *DRL* are usually with lager $Access_{cnt}$ and smaller $Page_{num}$, thus gaining a higher priority to be kept in cache, compared with the request blocks held in *IRL*.

Furthermore, *Req-block* supports downgraded merging of cached blocks and evicts them in batch, and Figure 6 demonstrates the workflow. As seen, the tail node of *DRL* is selected to be evicted as it has the least value of $Freq_{discnt}$. Then, we will merge it with the neighbouring data pages in the lower level list of *IRL* by referring to spatial locality, and evict them together in batch.

3.4 Implementation Details

Algorithm 1 demonstrates the specifications of *Req-block* in cache management with three-level linked lists. Note that, the cache of SSD devices is mainly used as a write buffer to cache the write data, so that only the data pages associated with write requests will be inserted to the cache.

As seen, *Lines 1-6* and *Lines 7-14* show the details about the creation of request blocks and the selection of victim blocks. The main routine of *Req-block* begins in *Line 15*. Specifically, *Lines 19-28* mainly illustrate the adjustment of request block when a hit on a small or large request block. *Lines 30-39* deal with the case of cache miss, in which it needs to evict the cached blocks to make room for the new data.

4 Experimental Evaluation

4.1 Experimental Setup

We have performed trace-driven simulation with *SSDsim* [26], which has been modified to support our newly proposed *Req-block* cache management scheme. The experiments were conducted on a local ARM-based machine, which has an ARM Cortex A7 Dual-Core with 800MHz, 128MB of memory and runs 32-bit Linux. Experimental settings are presented in Table 1, mainly corresponding to [12] [27]. Besides, we further investigate the performance of *Req-block* with different scales of data cache. 128GB SSD devices are generally equipped with 128MB DRAM by the proportion of 1GB:1MB [12], and at least 100MB of which is used to storing the mapping table to support page-level FTL [13]. Specifically, we set the size of data cache varying from 16MB to 64MB for our 128GB SSD device, excluding the DRAM space used for holding the mapping table.

To make a comprehensive assessment of *Req-block*, we select 6 commonly used disk traces that are collected from different kinds of applications. Specially, the 2016021613-LUN0 (labeled as *lun1*) is recently collected from the enterprise virtual desktop infrastructure(*VDI*) [28]. The remainder 5 traces are selected from the block I/O trace collection of Microsoft Research Cambridge [29]. The specification on the selected traces is shown in Table 2. In the table, **Frequent R** means the ratio of addresses requested not less than 3, and **(Wr)** implies the percent of write addresses in which.

Table 1. Experimental settings of *SSDsim*

Parameters	Values	Parameters	Values
Capacity	128GB	*Read latency*	0.075ms
Channel Size	8	*Write latency*	2ms
Chip Size	2	*Erase latency*	15ms
Page per block	64	*Transfer (Byte)*	10ns
Page Size	4KB	*GC Threshold*	10%
FTL Scheme	Page level	*DRAM Cache*	16/32/64MB

Table 2. Specifications on traces (ordered by the write ratio)

Traces	Req #	Wr Ratio	Wr Size	Frequent R(Wr)
hm_1	609312	4.7%	20.0KB	46.1%(83.9%)
lun_1	1894391	33.2%	18.6KB	12.4%(12.8%)
usr_0	2237889	59.6%	10.3KB	52.9%(32.9%)
src1_2	1907773	74.6%	32.5KB	79.6%(39.1%)
ts_0	1801734	82.4%	8.0KB	43.0%(58.1%)
proj_0	4224525	87.5%	40.9KB	62.5%(59.9%)

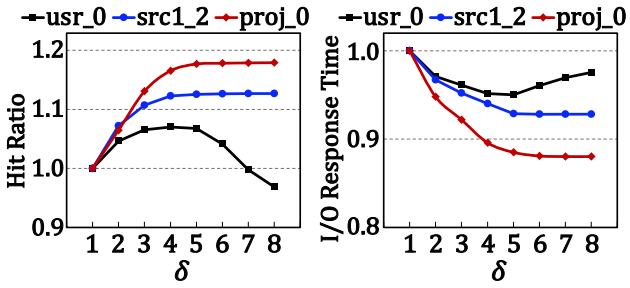

Figure 7. The sensitivity of parameters on hit ratio and I/O response time with 32MB cache. (Normalized to δ = 1).

Apart from the widely used *LRU* algorithm, we also take the *VBBMS* and *BPLRU* into comparison evaluation. The main design of the *VBBMS* and *BPLRU* schemes are summarized as follows:

- **BPLRU** [15] supports block granularity cache management. A block of cached pages is flushed onto physical SSD block of underlying flash memory on a replacement process. Furthermore, the cached block can be adjusted to the tail of the LRU list for the preferential eviction, while it is written sequentially.
- **VBBMS** [16] splits the cache space into a random request service region and a sequential request service region, by following the proportion of 3:2, and the sizes of virtual block in two regions are respectively set to 3 pages and 4 pages.

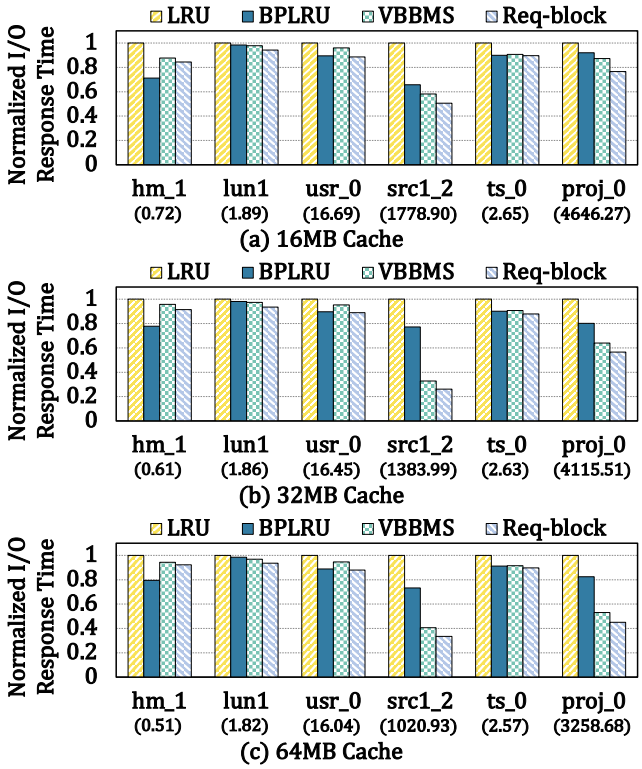

Figure 8. Comparison of overall I/O response time(Normalized to *LRU*). *Note that the numbers under the X-axis are the absolute values of LRU (unit: ms).*

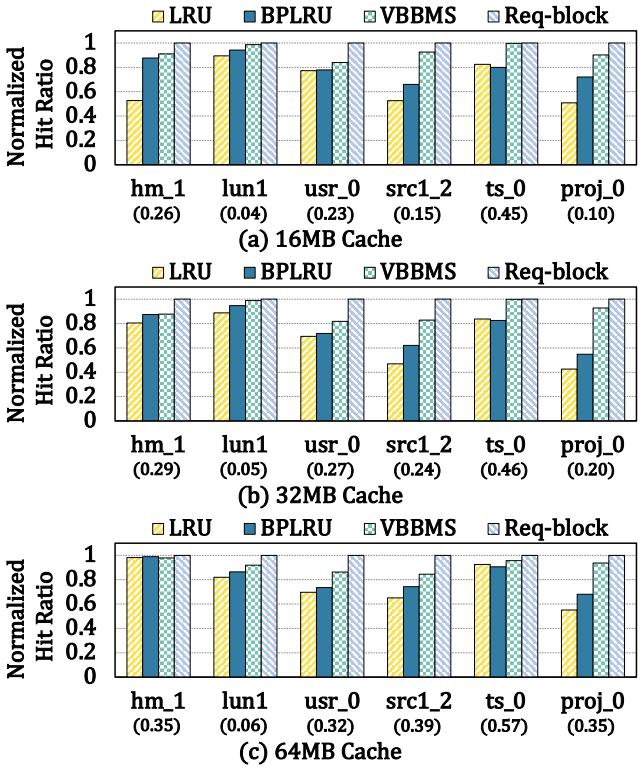

Figure 9. Comparison of hit ratio of varied cache configurations(Normalized to *Req-block*). *Note that the numbers under the X-axis are the absolute values of Req-block.*

We argue that *BPLRU* and *VBBMS* are the most related works of our proposal, as both of them have the consideration of (large size) sequential writes have less temporal locality.

4.2 Results and Discussion

4.2.1 Sensitivity of Parameters. In the *Req-block* scheme, the size limit of the request block in *Small Request List* (i.e. δ) is a sensitivity parameter, which is used to distinguish small and large request blocks. Generally, *Req-block* performs page-based cache management in *SRL* if δ is equal to 1, and a large δ implies request blocks that can be upgraded to the *SRL* list, may have more data pages.

That is to say, it is important to find a suitable δ for yielding an effective separation of request blocks. From the results of our comprehensive sensitivity tests, as shown in Figure 7, we conclude that a δ value of 5 results in a better hit ratio and I/O performance for most of the selected I/O workloads, thus we use this value by default in our evaluation.

4.2.2 I/O Response Time. I/O response time is the primary indicator of performance in SSDs. Figure 8 shows the normalized results of I/O response time after replaying the selected traces. We can see that *Req-block* can make a significant reduction on I/O response time. Specifically, it reduces I/O time by 23.8%, 11.3%, and 7.7% on average, in contrast to *LRU*, *BPLRU*, and *VBBMS* respectively.

VBBMS and *Req-block* result in better I/O performance with the selected I/O traces, in contrast to *LRU*. That is because *VBBMS* and *Req-block* perform cache evictions with a virtual block or request block granularity that contain several data pages, which exploits the multiple channels inside SSDs, and each eviction operation can make more available cache space.

BPLRU generally performs worse than *VBBMS* and *Req-block*, though it also manage the cached data with the block granularity. The reason is that *BPLRU* flushes the whole block consisting of many data pages onto a single SSD block, which fails to utilize access parallelism across multiple channels inside SSDs, and then affects I/O performance. Another interesting clue is about *BPLRU* works the best in the case of *hm_1*. We argue that flushing a block data onto a specific SSD channel only delays I/O processing at the same channel, which does not affect processing other I/O requests targeting at other channels, so that *BPLRU* may have better I/O performance when running read-intensive workloads.

We argue that the optimization of I/O response time is related to not only the cache hit ratio (see Section 4.2.3), but also the utilization of single or multiple channels in the contexts of SSDs (see Section 4.2.4).

4.2.3 Cache Hit. The term of cache hit means the ratio of the pages from the I/O request that is absorbed by the cache,

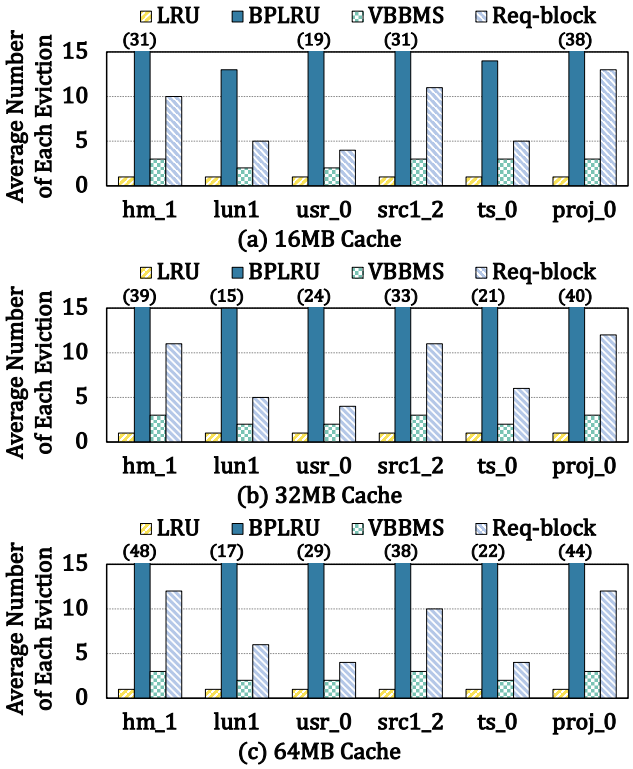

Figure 10. Comparison of average page number of each eviction.

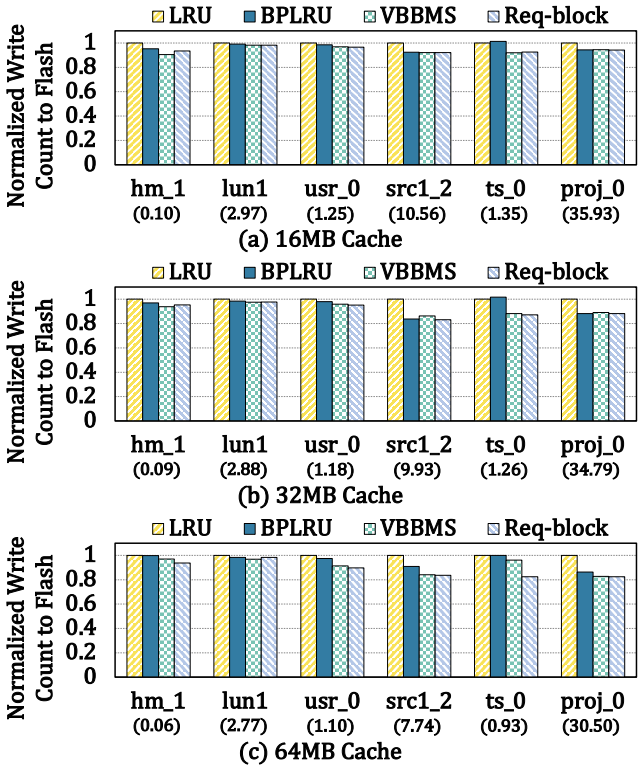

Figure 11. Comparison of write count to flash memory (unit: 10^6).

without accessing to underlying flash memory. A higher cache hit ratio implies less access to underlying flash arrays, which contributes to better I/O performance.

Figure 9 reports the normalized results of hit ratio after running the benchmarks with varied cache management schemes. As seen, the proposed *Req-block* scheme delivers the highest hit ratio compared to the other algorithms. Specifically, *Req-block* improves cache hit by 42.9%, 23.6%, and 4.1% on average, in contrast to *LRU*, *BPLRU*, and *VBBMS*.

Furthermore, *Req-block* achieves more pronounced improvements in case of replaying the traces having considerable number of both large and small requests, as *Req-block* can make a better separation of request blocks and the small request blocks are designated a high priority to be kept in the cache which contributes to better cache hits. For example, in the cases of *src1_2* and *proj_0*, our proposal of *Req-block* can improve the hit ratio by up to 100.0%, in contrast to the baseline of *LRU*. Similarly, *BPLRU* preferentially evicts blocks of large size sequential write and *VBBMS* employs three-fifth of the total cache capacity for buffering small size random requests, so that they can also yield better cache hits after replaying these traces compared to *LRU*.

Another observation is that *BPLRU* performs even worse on *ts_0* with respect to the measure of cache hits, compared to the baseline of *LRU*. This is because most of the requests in these traces are less than 3 data pages but the size of a block in *BPLRU* is 64 pages. Thus, the hot and cold level of the pages belonging to the same block can be uneven, which can cause the lower cache utilization.

4.2.4 Batch Eviction. To characterize the eviction in the three cache management schemes, we record the number of ejected pages in each eviction operation. Statistics are shown in Figure 10. As seen, our method of *Req-block* evicts pages in each eviction process between *VBBMS* and *BPLRU*. This is because *Req-block* evicts the data as the granularity of request block, while *BPLRU* and *VBBMS* evict with a block or virtual block granularity.

We emphasize that evicting more data pages (more than the number of channels in SSDs) does not directly contribute to better I/O performance, a large number of evicted pages may lead to flushing congestion and even an I/O performance degradation. Nowadays, high density SSDs usually come with multiple channels [30, 31]. For example, the *SM8266* controller enables 16 SSD channels and is designed to support the latest 3D TLC and QLC NAND flash technologies [32]. Thus, it is beneficial to exploit the parallelism of SSD channels for better I/O performance in cache management.

Three block-level cache management schemes, i.e. *BPLRU*, *VBBMS*, and *Req-block* eject multiple data pages in each eviction, that means the data pages are flushed to the flash memory in batch and more cache space can be freed up for upcoming write requests. However, this may intuitively increase

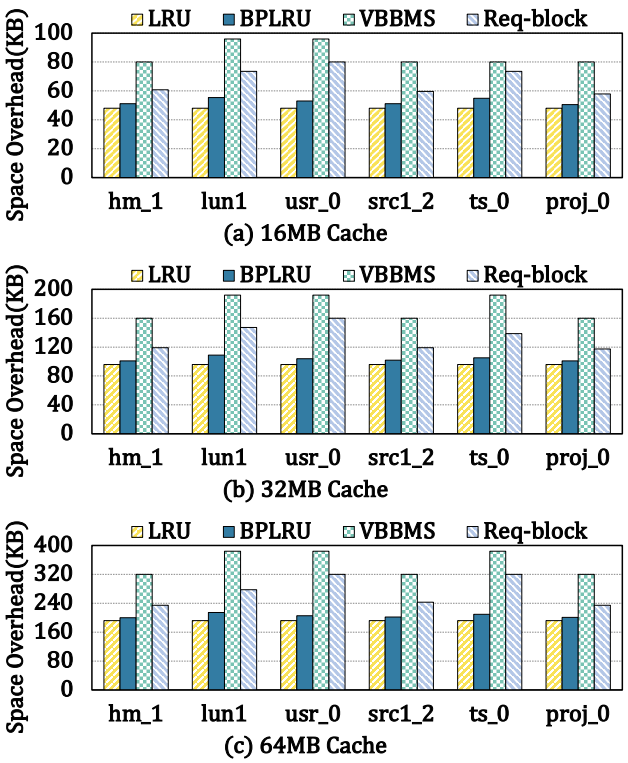

Figure 12. Space overhead of three cache management schemes.

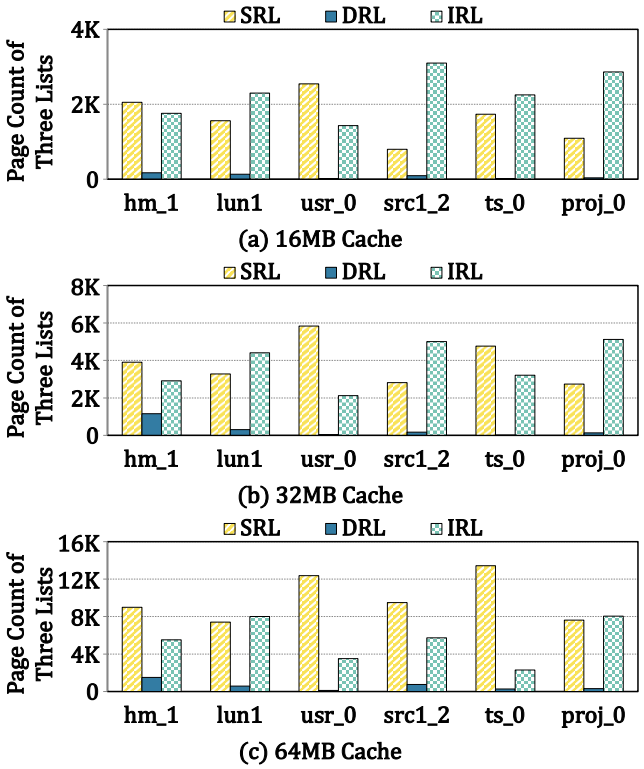

Figure 13. Page count of three lists in *Req-block*, which is logged once for every 10,000 requests (*K* means 1024).

the number of write operations to the flash memory if the flushed data pages are updated in the near future. To prove batch eviction of our proposal does not lead to more flush operations to flash array in the end, we record the write count after running the selected benchmarks, by using different cache management schemes.

As reported in Figure 11, *Req-block* causes the least number of write count in most of traces. More exactly, it can cut down the number of write count by 8.6%, 4.3%, and 1.1% on average, in contrast to *LRU*, *BPLRU*, and *VBBMS*. In fact, evicting more cold data pages earlier can make more room for hot data and thus boost I/O performance. Our proposal of *Req-block* preferentially ejects cold large request blocks, as the data pages of these request blocks are less likely to be updated frequently. On the other hand, *Req-block* can then hold more hot data pages in cache to absorb write requests and thus yield better performance.

4.2.5 Run-Time and Space Overhead. The time overhead of used cache management schemes is mainly caused by search and adjustment operations on the linked lists that manage the data structure of cached items. The search operations of cached pages in SSDsim simulator are performed on an AVL-tree with O(log n) time complexity, and n is the number of cached data pages. Besides, the adjustment operations on the linked-list cause O(1) time complexity. Overall, we conclude that the time complexity of Req-block is O(log n). In other words, the run-time overhead of *Req-block* is acceptable.

The space overhead is due to storing the linked lists, which is directly related to the number of cached items and the size of the list node that corresponds to a cache item. Specifically, the granularity of cached items in *LRU*, *BPLRU*, and *Req-block* is a page, a block, and a request block, and the corresponding node requires 12 Byte, 24 Byte, and 32 Byte, respectively. Specially, the *VBBMS* adopts a virtual block, which needs the same memory as a block.

Figure 12 shows the space overhead of three cache management schemes. As seen, our proposal of *Req-block* needs 67.6 KB, 133.6 KB, and 271.6 KB on average with different sizes of cache configurations, and corresponds to an average of 0.41% of total cache space. Besides, *LRU*, *BPLRU*, and *VBBMS* separately need 0.29%, 0.32%, and 0.53% of total cache space on average. Thus, we conclude that the space overhead of *Req-block* is comparable to that of *LRU*, *BPLRU*, and *VBBMS*.

4.3 Analysis on Lists of *Req-block*

This section characterizes the number of elements in three-level linked lists, adopted by the proposed *Req-block* scheme. Figure 13 reports the number of pages in three linked lists after replaying the selected traces. As shown, a majority of data pages in the cache is reserved in *SRL* and *IRL*, especially,

SRL contains the largest number of cached pages in most cases. This is because the request blocks in *SRL* are more likely with a higher *access count* and a lower *page number* and thus they have more opportunities to be kept in the cache. In addition, the newly inserted pages are initially added to *IRL*, so that the number of cached pages accounts for a reasonable proportion.

Another interesting observation shown in the figure is that *DRL* holds a small part of cached request blocks. This information further verifies that large block requests and their parts of data pages will be re-accessed with a low probability. Thus, managing the cached data as the request granularity and preferably keeping small requests in the cache can contribute to I/O performance improvements, which has been illustrated in Section 4.2.2.

5 Conclusion

This paper has proposed a request-granularity based cache management approach for SSD devices, called *Req-block*. Considering that the data of small sized requests are more likely to be accessed in the DRAM cache compared to those with larger size request data, *Req-block* directly manages the cached data in batches (i.e. request blocks) corresponding to the write requests. To this end, it supports three-level linked list and performs different cache management for the cached request blocks corresponding to write requests with different sizes. The scheme preferentially keeps small request blocks in the DRAM cache.

Through a series of simulation tests based on several real-world disk traces, we have demonstrated that our proposal enhances cache hits by up to `90.5%`, and reduces I/O latency by `14.3%` on average, compared to other state-of-art cache management schemes in SSDs.

Acknowledgments

This work was partially supported by "National Natural Science Foundation of China (No. 61872299, No. 62032019)", "the Natural Science Foundation Project of CQ CSTC (No. cstc2021ycjh-bgzxm0199)", and "Chongqing Graduate Research and Innovation Project (No. CYS22215)".

References

[1] J. Liao, F. Zhang, L. Li, and G. Xiao. Adaptive Wear-Leveling in Flash-Based Memory. In *IEEE CAL*, 2015.
[2] B. Kim, J. Choi, and S. Min. Design tradeoffs for SSD reliability. In *FAST*, 2019.
[3] J. Liao et al. A flexible I/O arbitration framework for netCDF-based big data processing workflows on high-end supercomputers. In *CCPE*, 2017.
[4] C. Liu, et al. LCR: Load-Aware Cache Replacement Algorithm for Flash-Based SSDs. In *NAS*, 2018.
[5] R. Micheloni. Solid-State Drive (SSD): a nonvolatile storage system. In *Proc. IEEE*, 2017.
[6] Z. Shen, L. Han, C. Ma, Z. Jia, T. Li, and Z. Shao. Leveraging the Interplay of RAID and SSD for Lifetime Optimization of Flash-Based SSD RAID. In *TCAD*, 2020.
[7] Cosmos OpenSSD Platform. http://www.openssd-project.org.
[8] X. Chen, Y. Li, and T. Zhang. Reducing Flash Memory Write Traffic by Exploiting a Few MBs of Capacitor-Powered Write Buffer Inside Solid-State Drives (SSDs). In *TC*, 2019.
[9] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee. CFLRU: a replacement algorithm for flash memory. In *CASES*, 2006.
[10] H. Chen, et al. ECR: Eviction-cost-aware cache management policy for page-level flash-based SSDs. In *CCPE*, 2021.
[11] H. Sun et al. Co-Active: A Workload-Aware Collaborative Cache Management Scheme for NVMe SSDs. In *TPDS*, 2021.
[12] Jun Li et al. Pattern-Based Prefetching with Adaptive Cache Management Inside of Solid-State Drives. In *ACM TOS*, 2022.
[13] W. Kang et al. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases In *SIGMOD*, 2014.
[14] J. Wan et al. DEFT-cache: A cost-effective and highly reliable SSD cache for RAID storage. In *IPDPS*, 2017.
[15] H. Kim and S. Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *FAST*, 2008.
[16] C. Du et al. VBBMS: A novel buffer management strategy for NAND flash storage devices. In *TCE*, 2019.
[17] W. Zhang et al. Improving Overall Performance of TLC SSD by Exploiting Dissimilarity of Flash Pages. In *TPDS*, 2020.
[18] S. Tripathy, and M. Satpathy. SSD internal cache management policies: A survey. In *Journal of Systems Architecture*, 2022.
[19] H. Jo et al. FAB: Flash-aware buffer management policy for portable media players. In *TCE*, 2006.
[20] J. Ou, J. Shu, and Y. Lu. A high performance file system for non-volatile main memory. In *Eurosys*, 2016.
[21] J. Hu et al. PUD-LRU: An erase-efficient write buffer management algorithm for flash memory SSD. In *MASCOTS*, 2010.
[22] Y. Chai et al. WEC: Improving Durability of SSD Cache Drives by Caching Write-Efficient Data. In *TC*, 2015.
[23] K. Zhou et al. Efficient SSD Cache for Cloud Block Storage via Leveraging Block Reuse Distances. In *TPDS*, 2020.
[24] D. Lee et al. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *SIGMETRICS*, 1999.
[25] S. Lee, D. Shin, Y. Kim, and J. Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. In *ACM SIGOPS Oper. Syst. Rev.*, 2008.
[26] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and C. Ren. Exploring and exploiting the multilevel parallelism inside SSDs for improved performance and endurance. In *TC*, 2013.
[27] C. Gao et al. Constructing large, durable and fast SSD system via reprogramming 3D TLC flash memory. In *MICRO*, 2019.
[28] C. Lee et al. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *SYSTOR*, 2017.
[29] D. Narayanan, et al. Write off-loading: Practical power management for enterprise storage. In *ACM TOS*, 2008.
[30] L. Zuolo et al. Memory driven design methodologies for optimal SSD performance. In *Inside Solid State Drives (SSDs)*, 2018.
[31] Y. Kang, Y. Jo, J. Cha, W. D. Bae, W. Lee, and S. Kim. FORESEE: An Effective and Efficient Framework for Estimating the Execution Times of IO Traces on the SSD. In *TC*, 2020.
[32] Silicon Motion. Launches Complete 16-Channel PCIe 4.0 NVMe Turnkey Enterprise SSD Controller Solution.

https://ir.siliconmotion.com/news-releases/news-release-details/silicon-motion-launches-complete-16-channel-pcie-40-nvme-turnkey/, 2020.