Visibility graph-based cache management for DRAM buffer inside solid-state drives

Zhibing Sha, Jun Li, Fengxiang Zhang, Min Huang, Zhigang Cai, François Trahay, Jianwei Liao

Visibility Graph-based Cache Management for DRAM Buffer Inside Solid-State Drives

ZHIBING SHA, JUN LI, FENGXIANG ZHANG, MIN HUANG, and ZHIGANG CAI, Southwest University of China, China

FRANCOIS TRAHAY, Télécom SudParis, France

JIANWEI LIAO, Southwest University of China, China

Most solid-state drives (SSDs) adopt an on-board Dynamic Random Access Memory (DRAM) to buffer the write data, which can significantly reduce the amount of write operations committed to the flash array of SSD if data exhibits locality in write operations. This paper focuses on efficiently managing the small amount of DRAM cache inside SSDs. The basic idea is to employ the visibility graph technique to unify both temporal and spatial locality of references of I/O accesses, for directing cache management in SSDs. Specifically, we propose to adaptively generate the visibility graph of cached data pages, and then support batch adjustment of adjacent or nearby (hot) cached data pages by referring to the connection situations in the visibility graph. In addition, we propose to evict the buffered data pages in batches by also referring to the connection situations, to maximize the internal flushing parallelism of SSD devices without worsening I/O congestion. The trace-driven simulation experiments show that our proposal can yield improvements on cache hits by between `0.8%` and `19.8%` , and the overall I/O latency by `25.6%` on average, compared to state-of-the-art cache management schemes inside SSDs.

CCS Concepts: • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Reliability**;

Additional Key Words and Phrases: Solid-state drives, Cache Management, Temporal and Spatial Locality, Visibility Graph, Batch Adjustment and Eviction

1 INTRODUCTION

The NAND flash memory-based solid-state drives (SSDs) are emerging to be the dominant storage devices for embedded systems, personal computers, and high performance platforms, because of their small size, high performance, random-access and low energy consumption [1, 2]. Apart from NAND flash arrays that permanently hold the data, an SSD device commonly has a faster but small amount of dynamic random access memory (DRAM) that acts as a cache for I/O operations. For instance, the *Silicon Armor SP A80* SSD is equipped with `512GB` flash array and `8MB~480MB` cache [3].

In general, the DRAM cache of SSD is utilized to not only keep logic-physical address mapping data structures, but also temporarily buffer the contents of overwrite or write requests [4, 5]. Then, the write requests can be quickly acknowledged after their contents are buffered in the cache, while the slow writes to flash arrays will be performed in the background [4]. As a result, it can greatly reduce the number of flush operations onto the underlying flash array, and thus improve the I/O performance and the lifetime of SSDs [6, 7].

Once the cache space becomes full, the cache management scheme must evict some buffered data and flush them to the underlying flash array, for making room for new data [7]. Thus, cache management significantly impacts the I/O performance because of the limited cache capacity in SSDs [8]. We understand that locality of reference characterizes

This is a revised version. A preliminary version of this work was published in the Proceedings of 2022 Design, Automation & Test in Europe Conference & Exhibition (DATE'22). Corresponding author: J. Liao, liaojianwei@il.is.s.u-tokyo.ac.jp. He works for College of Computer and Information Science, Southwest University of China, and State Key Lab. for Novel Software Technology, Nanjing University, P.R. China.
Authors' addresses: Zhibing Sha; Jun Li; Fengxiang zhang; Min Huang; Zhigang Cai, Southwest University of China, Chongqing, China, 400715; Francois Trahay, Télécom SudParis, Evry, France, 91011; Jianwei Liao, Southwest University of China, Chongqing, China, 400715.

the ability to predict future accesses from the past accesses, and is the base of cache management. There are two main types of locality: ***temporal*** and ***spatial***. Temporal locality refers to repeated accesses to the same data, and spatial locality refers to adjacent accesses to the nearby data, within short time periods [9].

Least recently used (*LRU*) is the most commonly used cache management scheme due to the simplicity and adaptability [10]. It is constructed on the top of the temporal locality of reference, as it only analyzes very limited information on recency. Clean first least recently used (*CFLRU*) is a variation of *LRU* that additionally considers whether the cached data are modified or not [11]. Similarly, the least frequently used (*LFU*) algorithm follows the concept of factoring out locality from reference counts, and the cached data having the least access frequency during the recent period will be firstly evicted [12].

With respect to sophisticated cache management in SSDs, Sun et al. [7] proposed a collaborative active write-back cache management scheme, which is collaboratively aware of I/O access patterns and the idle status of flash arrays (e.g. flash chips), to minimize the negative impacts of cache evictions. Wang et al. [5] introduced a cache management scheme for SSDs, with consideration of the access frequency of the buffered pages. They used the particle swarm optimization (*PSO*) technique [13] to predict the access frequency of pages, in order to guide cache evictions. Megiddo et al. [14] proposed adaptive replacement cache (*ARC*) algorithm. It dynamically balances recency and frequency by employing two *LRU* queues corresponding to the changes of access pattern, for directing cache replacement.

On the other hand, most workloads have a high spatial locality and temporal locality, and designing a cache that takes advantage of both locality of references can boost the storage performance in computing systems [15]. Du et al. [16] proposed a virtual block-based buffer management scheme for SSD devices, called *VBBMS* that groups the buffered data pages into virtual blocks according to their access patterns (i.e. random or sequential), and manages the buffered pages at the virtual block level. More importantly, Song et al. [17] proposed a new management scheme, called dual locality (*DULO*). It firstly balances both factors of temporal and spatial locality of workload, for directing cache management.

Although such sophisticated methods can improve the cache use efficiency in many specific cases compared to generally used *LRU* or *LFU*, they cannot cover all scenarios with expected I/O improvements [18]. Considering SSD devices are resource-limited, it is expected to integrate a simple and more versatile cache management scheme with such devices, to boost I/O performance more broadly. In this paper, we propose *VS-Batch+*, a cache management scheme for SSD devices that considers both temporal and spatial locality of references. To this end, *VS-Batch+* unifies both types of locality by using a visibility graph. In summary, it makes the following contributions:

- We propose to employ the ***visibility graph*** technique [19] for unifying both temporal locality and spatial locality of references in cache management of SSDs. Then, it uses four levels of linked lists that correspond to 4 connection cases of nodes in the visibility graph of cached data pages, for managing the pages in a differentiated manner.
- We present a batch-based promotion and demotion adjustment on the cached data pages. It promotes the hit pages, their neighboring pages and their (nearby) frequently accessed pages to high-level linked lists ***in batches***, and it demotes the cold data pages to low-level linked lists until they are evicted from the cache.
- We introduce adaptively ***refreshing the visibility graph*** of cached data pages, to provide real-time and effective information for cache management. It observes the number of evicted data pages that were members of the last-round of visibility graph, and triggers a new generation round if the number exceeds a predefined (theoretical) threshold.

- We design a ***batch-based eviction*** method to make more cache space in a round of eviction. It ejects not only the selected node in the eviction list, but also certain correlative data pages which are visible to the selected node and are accessed less frequently.

Note that, as an extension of our previous work [20], only the last 2 of contributions are new in this paper. As our experiments in Section 4 indicate, the newly proposed cache management scheme can further improve cache use efficiency and reduce the I/O latency in SSD devices.

The rest of the paper is organized as follows: Section 2 describes our motivation. Section 3 presents the proposed cache management scheme of *VS-Batch+*, that takes both temporal and spatial locality into account. Section 4 depicts the evaluation methodology and reports the experimental results. Section 5 summarizes the related work. Section 6 concludes the paper.

2 BACKGROUND AND MOTIVATIONS

2.1 Caching inside SSDs

Caching inside SSD can absorb certain overwrite and write requests to optimize SSD performance. Cache management mainly focuses on the replacement strategy to make room for new data by evicting some of the buffered data. Most of simple but effective cache replacement strategies are basically built on the top of temporal locality, such as *LRU*, *CFLRU*, or *LFU*. Moreover, existing advanced cache management approaches for SSDs, such as *PSO*, and *ARC*, only take the temporal locality of reference (or with the spatial factor) into account when carrying out cache replacement[1].

2.2 Visibility Graph

The technique of visibility graph has revealed its potential in describing the main characteristics of a time series [19]. This approach maps a time series[2] into a network, which reflects properties of the time series. In turn, the investigation of a network constructed from the time series through the visibility graph method, can reveal nontrivial information about the time series itself.

For constructing the visibility graph associated with a univariate time series recording values of a scalar observable x, this series is considered as a **two-dimensional** set of points (t_i, x_i) with $x_i = x(t_i)$. As defined in Equation 1, two of such points are regarded as being mutually connected vertices of the visibility graph if the convexity condition is fulfilled for all time points t_k with $t_i < t_k < t_j$ [19].

$$\frac{x_i - x_k}{t_k - t_i} > \frac{x_i - x_j}{t_j - t_i} \tag{1}$$

In fact, visibility graph and its variants have been broadly used in time series-relevant analysis, such as fault diagnosis of rolling bearings [22], wall turbulence analysis [23], and geophysical records analysis [24]. With respect to the storage domain, Tran et al. [21] disclosed that the sequence of block access on the storage server is ordered in time and can be split into successive parts by a constant time interval, meaning that the sequence resembles typical time series. Then, several work employed the technique of horizontal visibility graph (HVG) [25] to transform a time series of block access events to a connected network, for reflecting their spatial locality, and thus directing I/O optimization of data prefetching and merging requests [26, 27].

[1]Though *VBBMS* [16] and *BPLRU* [37] declare considering the factor of spatial locality by organizing buffered data pages as fixed-size virtual blocks, we think it is based on access patterns and fails to unify both temporal and spatial locality of references. Besides, the DULO cache management scheme [17] originally aims at hard disk drives, taking both locality of references into consideration.
[2]Time series data refer to sequences of data that are ordered either temporally, spatially or in another defined order [19, 21].

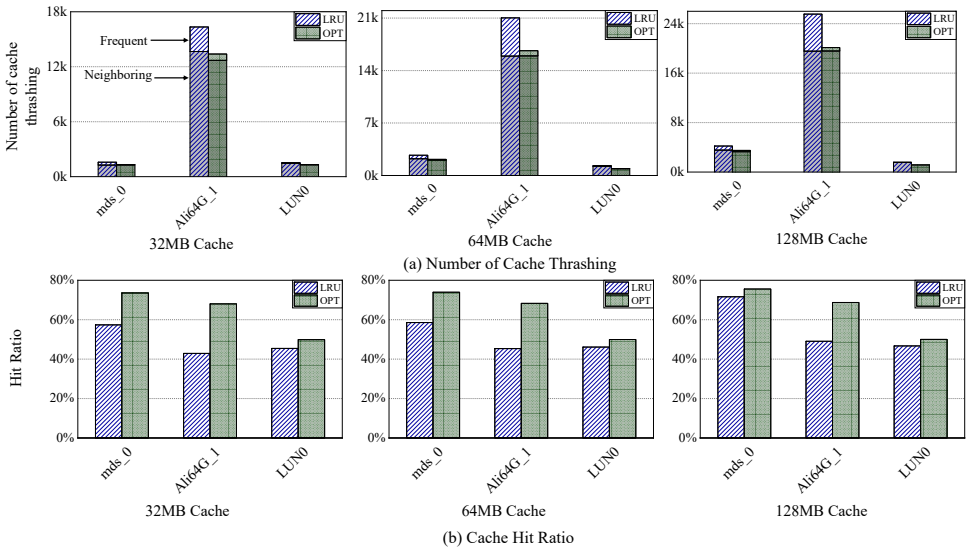


Fig. 1. Cache thrashing and its impacts after running some selected traces with *LRU* and *OPT*. (a) The number of cache thrashing events. The labels of *Neighboring* and *Frequent* correspondingly represent the thrashed items are either neighbors (i.e. spatial locality) and nearby frequently accessed data pages (i.e. spatial and temporal locality). (b) The results of cache hit ratios.

2.3 Motivations

In order to verify whether the factor of spatial locality matters or not in cache management of SSDs, we have performed a series of trace-driven simulation experiments and collected the results. Section 4.1 describes in details the experimental platform and benchmarks. We first define the term of ***cache thrashing*** events as what happens when a specific data page is kept in the SSD cache, ***while*** its address (i.e. logical page number) neighboring or nearby frequently accessed cached item is evicted out and loaded into the cache again. After replaying the selected traces, we count the number of cache thrashing events and record the I/O performance, when using the commonly used *LRU* cache management scheme and the theoretically *OPT* replacement policy [12]. Note that we slightly modified the *OPT* replacement policy by purposely avoiding cache thrashing as much as possible, to quantify how much performance we can improve through reducing thrashing events.

As the results shown in Figure 1(a), the *LRU* policy generates between `1279` and `25544` cache thrashing events, taking up to `24.1%` of total evictions, after running the selected traces. On the other side, the *OPT* replacement scheme can significantly reduce the number of thrashing events, by `18.3%` on average, in contrast to *LRU*. Figure 1(b) presents the results of cache hit comparison between *LRU* and *OPT*. Compared to *LRU*, the *OPT* replacement scheme improves the cache hits by `26.0%` on average, which are directly caused by the reduction of cache thrashing when replaying the selected traces.

More importantly, we see that thrashing items consist of neighbors and nearby frequently accessed data pages to the in-cache data pages. Therefore, we argue that it is possible to avoid a cache thrashing if the thrashed data item is managed in a batch with its in-cache neighboring or nearby data items. In brief, the spatial locality of cached data pages is worth considering in cache management of SSDs, to boost cache use efficiency and I/O performance.

Such observation motivates us to build an efficient and adaptive approach on the top of **two-dimensional** locality of references (i.e. the temporal locality and the spatial locality), by using a unique data structure. As a result, the number of cache hits and the I/O performance of SSDs can be improved when running a wide range of applications.

3 VISIBILITY GRAPH-BASED CACHE MANAGEMENT

3.1 System Overview

The basic principle of our approach, called *VS-Batch+*, is to take both spatial locality and temporal locality into account, for guiding cache management in SSDs. Once a specific data page is hit in the cache or is newly loaded into the cache, *VS-Batch+* accordingly adjusts the cached data pages adjacent to it, as well as the hot access data pages near it in batches. Through reducing the number of cache thrashings, we can yield performance gains if the cached data are requested again shortly by following the spatial locality of reference.

To this end, we first use the technique of visibility graph [19] to unify both locality of references of the cached data pages. Next, we introduce four-level linked lists that help managing the different visible types of cached data pages, where each node in the lists corresponds to a buffered data page. After that, we can identify the hot buffered data with their neighboring data and nearby (hot) data, and preferably keep them in the SSD cache. Finally, we support batch-based cache eviction to make cache space in a round of eviction, by also referring to the connection situations in the visibility graph of cached pages.

3.2 Design Specifications on *VS-Batch+*

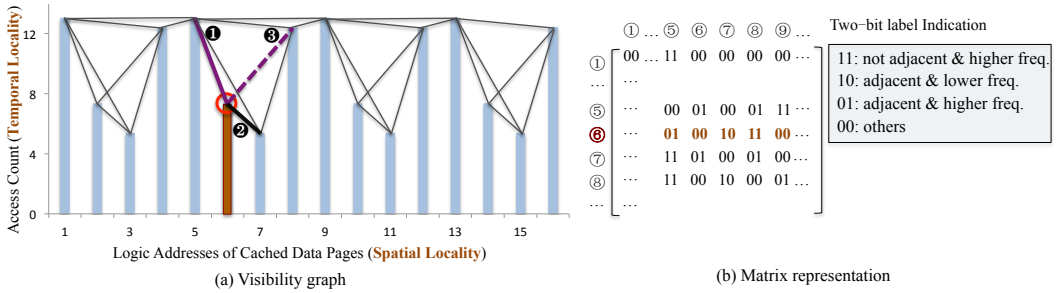
3.2.1 Unifying Temporal and Spatial Localities with Visibility Graph. In order to model the temporal and spatial localities of all cached data pages, we employ the visibility graph technique. Specifically, a sequence of logical page addresses of cached data pages can be transformed to a connected graph where each node represents a cached data page, and the node's value is set as the *access count* of the cached page in previous time windows. Two nodes in the visibility graph are connected by an edge if visibility exists, indicating it does not intersect any intermediate data height. It has the following visibility criteria: two arbitrary data values (x_a, y_a) and (x_a, y_b) have visibility, and thus become two connected nodes in the graph, if any other data (x_c, y_c) placed between them fulfills:

$$y_c < y_b + (y_a - y_b) \times \frac{x_b - x_c}{x_b - x_a} \tag{2}$$

where x_i indicates the sequential number of the ith node, and y_i means the access count of the ith node.

Figure 2 (a) illustrates an example of a visibility graph that is transformed from a given sequence of access counts of 16 cached pages. Given a specific node (for example *Node* #6 in Figure 2), we define three types of visibility: ❶ adjacent (the nearest neighbors) and higher access frequency (e.g. *Node #5*), ❷ adjacent and lower access frequency (e.g. *Node #7*), and ❸ not adjacent and higher access frequency (e.g. *Node #8*). Note that the adjacent node is defined as the node with a gap of no more than 4 to the given node, in term of logical page number.

More specifically, we regard the spatial locality and the temporal locality as the two-dimensional data, and employ visibility graph to integrate them. In other words, the adjacent nodes can see each other in the visibility graph, reflecting the spatial locality of reference. The current node can see other nodes having a high access frequency, addressing the temporal locality of reference. Consequently, when the current node is hit, we will adjust it with its "visible" data pages of a data page being accessed, according to the visibility type. The adjacent nodes (i.e. *Node #7*) can be seen by *Node #6*, reflecting the spatial locality of reference. In addition, *Node #6* can see *Node #8* that has a larger access count, even



(a) Visibility graph (b) Matrix representation

Fig. 2. The visibility graph of access frequency of 16 cached data pages. (***a***) visibility graph. *The marked Node (#6) has three kinds of connected (visible) nodes:* ❶*adjacent and higher access frequency,* ❷*adjacent and lower access frequency, and* ❸*not adjacent and higher access frequency.* (***b***) matrix representation of visibility graph. *Two-bit value implies the connection case.*

though they are not address-neighboring, reflecting the temporal locality of reference. Moreover, we understand that *Node #6* can see *Node #5*, reflecting both spatial and temporal locality of references. Thus, we can achieve the goal of decreasing cache thrashing events if the thrashed data pages are managed in batches with their in-cache neighboring or nearby data pages, by resorting to the visibility graph.

The visibility graph is stored as a matrix that depicts the connection between couples of buffered data pages, as shown in Figure 2(b). The two-bit value element implies the corresponding visibility situation between two nodes in the visibility graph. Once a buffered page is hit, we check the matrix to locate its visible nodes and adjust them accordingly. Section 3.2.3 will depict the specifications on node adjustment. Besides, we support batch evictions by also referring to the visibility graph when there is not enough cache space, and the details will be illustrated in Section 3.2.4

3.2.2 Visibility Graph Reconstruction. Refreshing the visibility graph of buffered data pages is beneficial to cache management on the fly. Our previous work of *VSBatch* [20] adopts periodical reconstruction while the amount of write data in the new round reaches the size of cache. We argue this is not the best way ensuring the effectiveness of visibility graph generation. On the one side, it does not rebuild the visibility graph of buffered pages until the refresh cycle is reached, even though a considerable number of buffered pages are evicted since the last round of visibility graph generation. On the other side, it must reconstruct the visibility graph after the cycle is reached, even though not many pages are evicted from the cache and the current graph can still reflect the visibility situations of the majority of cached pages.

To address the aforementioned issue, this section discusses our proposal of visibility graph refreshing by considering the changes of visibility cases in the graph. The amount of connection situations of cached pages is reflected in the number of edges in the visibility graph. The maximum number of connections in a n-node visibility graph must follow Equation 3.

$$\binom{n}{2} = \frac{n \times (n-1)}{2} \tag{3}$$

The number of cached pages in the original visibility graph keeps decreasing, as the cache is continuously updated with page evictions. The authenticity of the visibility graph is maintained until the number of removed connections reaches half the theoretical maximum by referring to "if G has m edges, then G can be made bipartite by removing at most $m/2$ edges [28]".

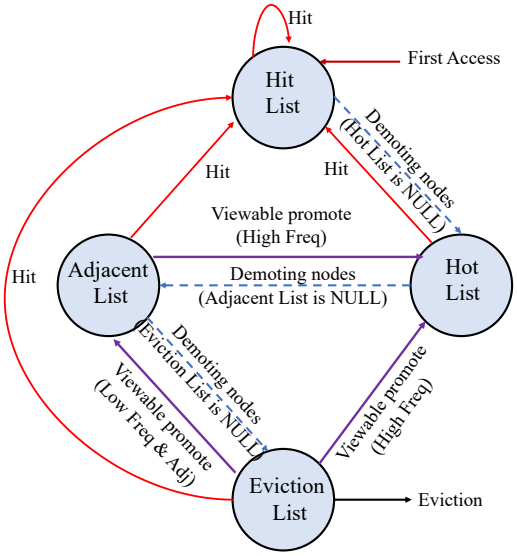


Fig. 3. Diagram of node transfers in four-level linked lists, triggered by different operations.

When the number of cached pages (i.e. the nodes used to build the visibility graph) decreases to $(\frac{n}{\sqrt{2}})$, that indicates `29.3%` of cached nodes have been evicted since last round of visibility graph generation, a new round of reconstruction must be forcibly triggered. Reconstructing the visibility graph more frequently can improve the direct cache management, but it incurs more overhead and thus affects I/O processing.

Section 4.2.1 carries out a case study by using different ratios that are less than the theoretical upper limit of `0.293`, to disclose the impact of varied thresholds for refreshing the visibility graph of cached pages.

3.2.3 Batch Adjustment in Cache. In order to manage the data pages located in the cache, *VS-Batch+* maintains four-level linked lists: *Eviction list (level 3)*, *Adjacent list (level 2)*, *Hot list (level 1)*, and *Hit list (level 0)* with the descending order. Figure 3 illustrates the node transfers in four-level linked lists, triggered by different events. At the initialization stage (i.e. all lists are empty), all the nodes of cached data pages are linked in the *Eviction list* with the *LRU* fashion. Once a given cached page is hit again, *VS-Batch+* carries out the promoted batch adjustment, to move the nodes of hit page and their visible (cached) data pages into higher-level lists. The proposed *VS-Batch+* method evicts the buffered data page to make space for the new data, *if-and-only-if* its corresponding node is the tail of *Eviction list*. When a cached data page is accessed, *VS-Batch+* moves the corresponding node to the head of the *Hit list*, and adjusts relevant nodes that can be seen in the visibility graph. Specifically, the visible and high frequency access nodes are moved to the *Hot list*, and other adjacent nodes are moved to the *Adjacent list*. Moreover, *VS-Batch+* supports demoting all nodes of a specific linked list, to the neighboring low-level linked list if the low-level list becomes empty.

As the example in Figure 4(a) shows, once **Page *F*** is hit, its node is directly moved to the head of *Hit list*. Meanwhile, the nodes labeling with F_{11} and F_{01} are moved to the *Hot list* as they have a larger access count than **Page *F***, and the node labeling with F_{10} is moved to the *Adjacent list* as it is adjacent to **Page *F***, even though it has a smaller access

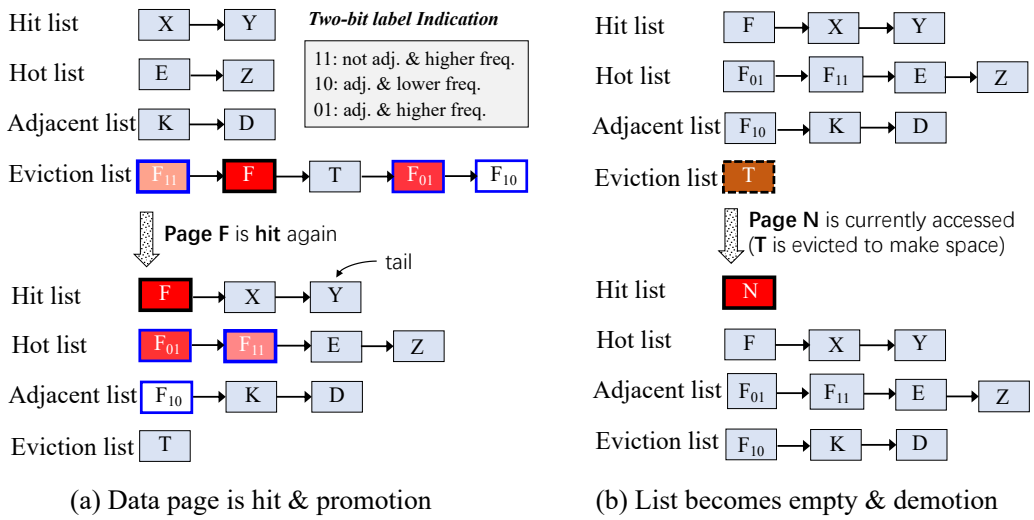


Fig. 4. Batch adjustment of cached items in *VS-Batch+* (assuming the SSD cache buffers only 11 data pages in total for illustration simplicity). In the node, the identifier with a subscript indicates the connection case in the visibility graph, from the view of the identifier page. Note that both the hit node or the newly inserted node will be placed on the head of *Hit list*.

count. In brief, *VS-Batch+* supports promoted adjustment of cached data, to protect the data pages whose neighbor data pages are recently requested.

On the other side, *VS-Batch+* supports hierarchical demoted moves when a lower-level list becomes empty. As illustrated in Figure 4(b), **Page *N*** is requested to be written to SSDs, but it is not in the cache and the cache is currently full. To service this request, *VS-Batch+* first evicts selected nodes of the *Eviction list* (e.g. **Page *T***). As the *Eviction list* becomes empty, all the nodes of other high level linked lists are moved downward step by step: the nodes of the *Adjacent list* move to the *Eviction list*, the nodes of the *Hot list* move to the *Adjacent list*, etc. Finally, **Page *N*** is loaded into the cache, and its node is inserted as the new head of the *Hit list*.

3.2.4 Cache Eviction. The evicted data pages are flushed onto flash cells of SSD channels, to make cache space for the new data. In order to maximize flushing parallelism on SSD channels, *VS-Batch+* supports evicting the cached data pages in batches, by also referring to the visibility connections. When the tail node of the *Eviction list* is selected to be ejected, it also selects relevant nodes in the *Eviction list* that are **visible** from the tail node and have a smaller access count. Then, all selected data pages are evicted.

Furthermore, we limit the size of batch flushing requests to be less than the number of idle channels of SSD. Consequently, the ejected pages can be flushed onto SSD channels in parallel, without worsening I/O congestion on SSD channels.

3.3 Implementation Details

Algorithm 1 illustrates the details on batch adjustment and batch eviction on the nodes in four-level linked lists. *Lines 1-12* present batch adjustment of nodes in the lists, while the corresponding data page is hit. It moves the hit node to Level `0` of *Hit list*, and the relevant visible nodes to *Hot list* and *Adjacent list* accordingly.

ALGORITHM 1: Batch Adjustment and Eviction

```
1  Function vs_batch_move(node_addr)
2      /*obtain the set of page addrs of visible nodes*/
3      addr_set = get_addrs_from_vs(node_addr);
4      for each addr in addr_set do
5          /*obtain the level of original list of addr*/
6          ori_level = get_node_level(addr);
7          /*obtain the dest list of addr with visibility type*/
8          vs_t = get_vs_t(addr, node_addr);
9          dst_level = get_dst_level(addr, vs_t);
10         if ori_level ≤ dst_level then
11             /*move to the head of destination list*/
12             move_to_list(addr, dst_level);
13 Function vs_batch_evict(node_addr, idle_chs)
14     int cnt = 0;
15     evict_from_list(node_addr);
16     addr_set = get_addrs_from_vs();
17     /*evict adj. & low freq. nodes from Eviction list*/
18     for each addr in addr_set do
19         level = get_node_level(addr);
20         vs_t = get_vs_t(addr, node_addr);
21         /*limit # of evictions ≤ idle channels*/
22         if level == 3 and vs_t==10 and cnt++ ≤ idle_chs then
23             evict_from_list(addr);
```

Lines 13-23 depict the process of batch eviction. Apart from the selected eviction of data page, other relevant visible buffered pages (in *Eviction list*) that have fewer accesses than the eviction page will also be ejected. Moreover, we limit the number of batch evictions to less than the number of idle SSD channels at the current point, for avoiding I/O congestion.

4 EXPERIMENTS AND EVALUATION

4.1 Experimental Setup

We have performed trace-driven simulation with *SSDsim (ver*2.1*)* [29], which has been modified to support the newly proposed cache management scheme, on a local ARM-based machine. The machine has an ARM Cortex A7 Dual-Core with 800MHz and 128MB memory. Table 1 shows our settings of experiments, by mainly referring to [30, 31]. To further investigate how our proposal works with varied scales of SSD cache, we set the cache size varying from `32MB` to `128MB`. In the tests, the configured cache space is dedicated for the write data, and the mapping table is separately saved. The parameter *Refresh Threshold* represents the default threshold for triggering the visibility graph reconstruction: when the ratio of evicted data pages to the total cached pages becomes higher than this threshold (i.e. `0.2`), the visibility graph is reconstructed.

Considering GC is the unique nature of SSD devices, we aged the simulated SSD to `88.0%` of capacity (the available space is `12.0%` of capacity, closing to the GC threshold of `10.0%`) for triggering more GC operations. Consequently, we can check the effectiveness of our proposal in the scenarios of many GCs in SSDs.

Table 1. Experimental settings of *SSDsim*

Chip parameters	
(Die, Plane, Block, Page)	`(1, 4, 256, 256)`
(Page size, Cell density)	`(8KB, QLC)`
Read latency (LSB-...-MSB)	`(90, 120, 150, 180)us`
(Program latency, Erase latency)	`(1.3ms, 10ms)`
SSD parameters	
(channel, chip)	`(8,4)`
(FTL, GC trigger)	`(Page-level, 10%)`
Transfer time per byte	`5ns`
DRAM capacity	`32M/64M/128M`
DRAM access latency	`1us`
VS reconstruction threshold	`0.2`

We employed `six` widely used disk traces of real-world applications in our evaluation tests [30, 32]. Besides `two` traces from the trace collection of Microsoft Research Cambridge [33], we also selected `two` block I/O traces from a part of an enterprise virtual desktop infrastructure (VDI) [34]. Specifically, they are additional-01-1617-LUN4 (labeled as *LUN0*), and additional-01-1618-LUN3 (*LUN1*). The rest `two` recent traces come from *Alibaba Cloud* [35], corresponding to the first two hour traces of `64GB` virtual disk (same capacity of our emulated SSD device), labeled as *Ali64G_1* and *Ali64G_2*.

The detailed specifications on the used traces are shown in Tables 2 and 3. As seen in Table 2, **Freq R** means the ratio of addresses requested not less than 3, and **(Wr)** implies the percent of write addresses in which. Furthermore, we defined two correlation coefficients of **Temp** and **Spat**, to reflect spatial and temporal locality of the selected benchmarks, respectively. To be specific, **Temp** implies the degree of temporal locality, referring to the probability that a logical address of page will be accessed again in the next time window. **Spat** indicates the degree of spatial locality, referring to the probability that a logical address of page is requested in a time window and its adjacent logical address will be also accessed in the same time window. We employ 1-second time window configuration by referring to [36], when computing the measures of spatial and temporal locality of benchmarks. Table 3 presents the details on the intervals between two I/O requests in the selected trace, in Cumulative Distribution Function, to reveal the bursty level of I/O requests. Generally, a small value of interval means congested I/O workloads, and will cause a large average I/O latency for servicing a read/write request.

Apart from *LRU* and the proposed *VS-Batch+* method, the following `three` schemes are also implemented in comparison evaluation:

- *VBBMS* [16] considers both temporal and spatial factors and manages the cached data pages as the granularity of virtual block. Moreover, it refers to access patterns and divides the buffer into two regions for separately fulfilling random requests and sequential requests. The size of virtual block in the sequential region and the random region is `6` and `8` pages, respectively. We argue that *VBBMS* is the most related work of our approach.
- *Co-Active* [7] first classifies the data into hot and cold categories, by referring to the factor of temporal locality. Then, it proactively evicts the (cold) data pages from the cache if their destination (underlying) SSD channels are idle, for cutting down the wait time of flushing.

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

Traces	Req #	Wr Ratio	Wr Size	Freq R (Wr)	Temp	Spat
mds_0	1211034	88.1%	7.2KB	0.3%(49.8%)	23.0%	29.0%
Ali64G_1	2931271	85.2%	6.5KB	18.2%(32.5%)	21.6%	11.9%
Ali64G_2	2729850	84.8%	7.5KB	10.1%(17.5%)	20.3%	11.0%
web_0	2029945	70.1%	8.6KB	22.7%(71.3%)	13.0%	35.1%
LUN1	983607	49.2%	16.3KB	0.9%(4.5%)	4.7%	23.3%
LUN0	966916	32.5%	18.8KB	1.8%(5.2%)	1.0%	23.4%

Table 3. Cumulative distribution function of request intervals in the selected traces

Traces	20%	40%	60%	80%	100%
mds_0	4.8us	17.2us	49.4us	156.6us	287.2ms
Ali64G_1	17.0us	38.0us	274.0us	1746.0us	45.9ms
Ali64G_2	19.0us	44.0us	390.0us	1997.0us	48.6ms
web_0	1.7us	10.8us	36.0us	143.2us	298.4ms
LUN1	25.1us	350.0us	764.9us	2529.0us	539.3ms
LUN0	22.8us	97.8us	867.8us	3019.7us	918.3ms

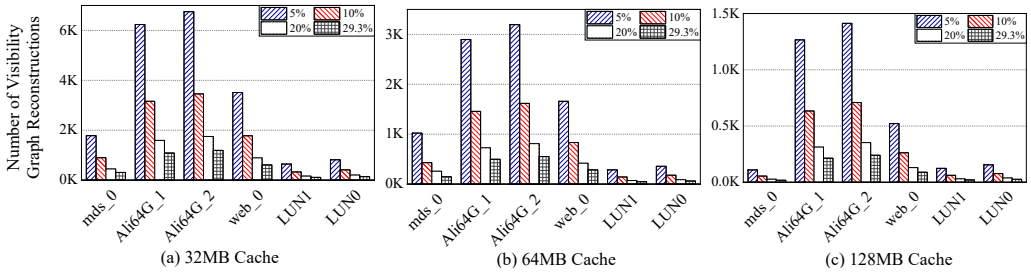


Fig. 5. The number of visibility graph reconstructions with varied reconstruction thresholds in *VS-Batch+*.

- *BPLRU* [37] supports the block-level granularity cache management. A block of cached pages is flushed onto a 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 has been filled to the full.
- *VS-Batch* [20] is our previous work, and its main contribution is to use the technique of visibility graph to unify both temporal locality and spatial locality of references. It periodically generates the visibility graph of buffered data pages, when the total amount of write data of I/O requests becomes greater than the size of cache, since the last processing round. Compared to *VS-Batch+*, *VS-Batch* does not support the features of adaptive reconstruction of visibility graph of cached data pages, and visibility graph-based batch evictions.

In all selected comparison counterparts of cache management, *LRU* is the typical cache scheme for conventional storage systems, *VBBMS*, *Co-Active*, and BPLRU are representative cache schemes for flash memory. More specially, *VBBMS* and *Co-Active* are fine-grained granularity schemes, and *BPLRU* is the coarse-grained granularity scheme for

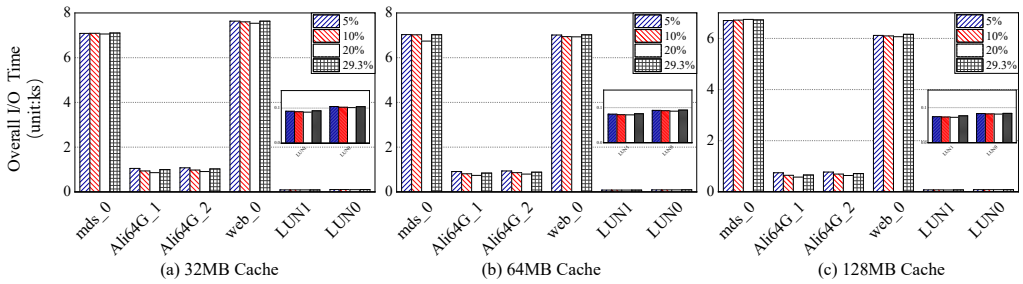


Fig. 6. The overall I/O latency with varied reconstruction thresholds of visibility graph in *VS-Batch+*.

cache management in SSDs. In both *VS-Batch* and *VS-Batch+*, we make use of the obtained visibility graph to direct cache management in the forthcoming time windows. Considering the range of spatial locality, we check `64` cached pages on the left and right of the vertical axis by referring to their addresses, when building the visibility connections for a given data page.

4.2 Sensitive Analysis and Statistics on *VS-Batch+*

4.2.1 Impact of the reconstruction threshold. The choice of the refreshing threshold of visibility graph of cached data pages implies an engineering trade-off. A small refreshing threshold triggers more reconstructions of visibility graph of cached pages, that can more exactly direct cache management but may increase I/O congestion. On the other side, a large refreshing threshold can reduce the number of reconstruction operations, but the visibility graph fails to reflect the connections of cached data pages in a timely manner. To further understand the impact of the value of the refreshing threshold, which is critical to the performance of the proposed method, we have replayed the traces to exploit its influence.

Figures 5 and 6 respectively present the number of visibility graph reconstructions and the overall I/O response time with different refreshing thresholds in the range of [`0.05`, `0.293`]. As seen, in the case of `0.05` (i.e., the lower limit threshold), *VS-Batch+* triggers more than several times the reconstructions, compared to the case of `0.2`. More visibility graph reconstruction can better direct cache management, but the reconstruction process delays processing on normal I/O requests, which must offset the benefits of frequent reconstructions of visibility graph. In the case of `0.293` (i.e., the upper limit threshold), we see *VS-Batch+* results in a slight decrease in visibility graph reconstructions, but the obsolete visibility graph of cached pages may fail to reflect the visibility connections of cached data in a timely manner and thus affects I/O performance.

We can see that refreshing threshold of `0.2` can yield the best results of I/O latency in most benchmarks, because it can well balance the overhead and the benefit of visibility graph reconstruction. As a result, we employ this value as the default value of the threshold of visibility graph reconstruction in our tests.

4.2.2 Statistics on Batch Adjustments and Evictions. To evaluate the batch adjustment and eviction feature, we record the number of node movements in each batch adjustment and the number of ejected data pages in each eviction, when replaying the selected traces with *VS-Batch+*. As seen in Figure 7 (a), *VS-Batch+* leads to certain batch adjustment operations, and each move to *Hit list* results in `1.6` moves to *Hot list* (higher frequency nodes) and `1.1` moves to

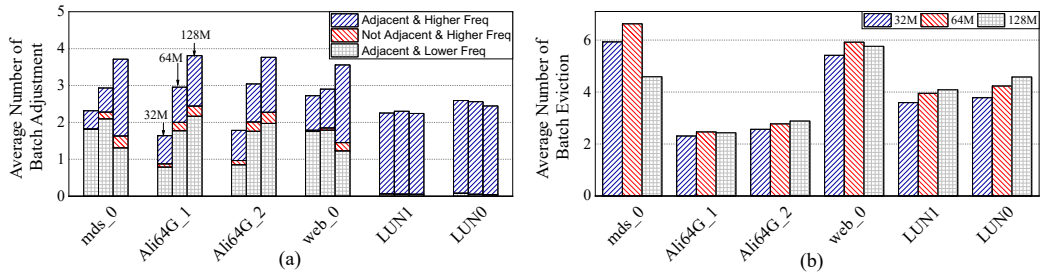


Fig. 7. Statistics on (a) batch adjustment, and (b) batch eviction of *VS-Batch+*.

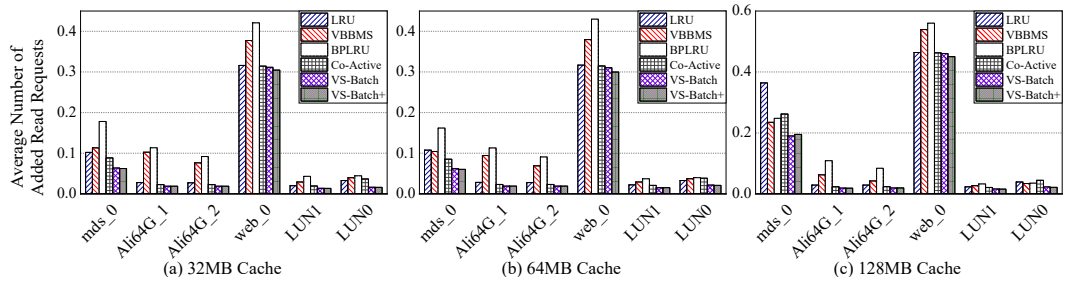


Fig. 8. The average number of blocked read requests after servicing a write request.

Adjacent list (adjacent and lower frequency nodes). This fact discloses that *VS-Batch+* does have many batch adjustment operations by also referring to spatial locality, which are the cause of the improvement on cache hits.

Figure 7 (b) reports the average number of ejected pages per eviction processes. We see that it ejects 4.1 cached data pages on average, that is smaller than the number of SSD channels in our configuration, in each eviction process, which can demonstrate *VS-Batch+* can well utilize the flushing parallelism of SSD channels and then boost I/O performance.

Furthermore, *VS-Batch+* ejects the cached data pages in batches by taking up multiple channels for each eviction, which seems to block more read requests and then affect parallelism of SSDs. In order to see if our mechanism of batch eviction really affects parallelism, we count the average number of newly inserted read requests when flushing the cached data pages in batches. Note that cache eviction does not block write requests, since it is caused by write transactions for making cache space.

Figure 8 presents the results, and it shows *VS-Batch+* does not increase the number of blocked read requests, in contrast to other schemes. This is because the enqueued read requests have higher priority than the enqueued write requests, batch eviction does not affect the enqueued read requests, and only delays the limited number of incoming read requests during flushing processes. More importantly, *VS-Batch+* can yield the least number of blocked read requests in some cases. This is because each round of batch eviction can make more cache space to avoid carrying out cache eviction for every write request (adopted by other cache polices), which benefits the reduction of the blocked read requests after satisfying all write requests.

4.2.3 Analysis on Visibility Graph Reconstruction. Contrary to our previous work of *VS-Batch* that periodically generates the visibility graph of cached data pages for directing cache management, *VS-Batch+* adaptively forms the visibility

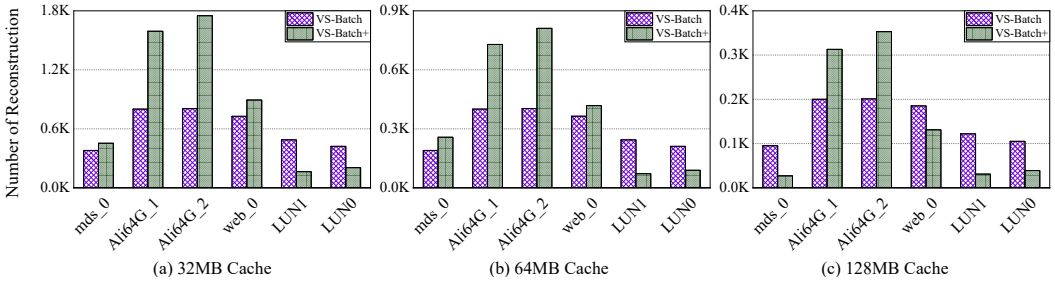


Fig. 9. Comparison of visibility graph reconstructions by using *VS-Batch* and *VS-Batch+*.

graph according to the number of evicted data pages. Figure 9 presents the results of visibility graph reconstructions when using both schemes.

On the one hand, *VS-Batch+* brings about less reconstructions of visibility graph by `63.9%` on average after running all *LUN* traces, in contrast to *VS-Batch*. Both *LUN* traces are read-heavy workloads and do not have intensive cache evictions, so that *VS-Batch+* results in less reconstructions of visibility graph. As a result, *VS-Batch+* can reduce the side effects on I/O processing caused by visibility graph reconstructions.

On the other hand, *VS-Batch+* generally leads to more visibility graph reconstructions by more than `43.6%` on average after replaying for write-intensive traces, comparing with *VS-Batch*. *VS-Batch+* triggers a round of reconstruction of visibility graph while the size of evicted data pages reaches the pre-defined ratio threshold to the whole cache. In other words, a larger cache size can absorb more write requests that can cut down the reconstruction frequency, and this is the cause of *VS-Batch+* increasing the I/O performance when the cache size becomes large.

Thus, we summarize that adaptive refreshing visibility graph according to the number of evicted data pages, can balance the overhead of visibility graph reconstruction and the visibility sit of in-cache data pages, for eventually reducing the I/O latency.

4.3 Performance Measurements and Discussions

4.3.1 Cache Hits and Thrashing. We first define the metric of the number of cache hits without flushing the buffered data onto underlying SSD cells. This term means the write contents can be directly saved in the cache without ejecting other buffered data. Then, the write request can be completed with a lower latency if its contents can be directly absorbed in the cache.

Figure 10 reports the cache hits ratio after running the benchmarks with varied cache management schemes, and the schemes of *LRU*, *VBBMS*, *BPLRU Co-Active*, *VS-Batch*, and *VS-Batch+* achieve the average hit ratios of `49.5%`, `58.1%`, `51.6%`, `56.6%`, `58.8%`, and `59.3%`, respectively. As illustrated, *VS-Batch+* performs the best, and achieves an improvement on cache hits by `19.8%`, `2.1%`, `15.0%`, `4.7%`, and `0.8%`, in contrast to *LRU*, *VBBMS*, *BPLRU*, *Co-Active*, and *VS-Batch*. To further analyze why *VS-Batch+* improves the cache hit ratio, Figure 11 reports the number of cache thrashing events, and the average numbers caused by the schemes of *LRU*, *VBBMS*, *BPLRU*, *Co-Active*, *VS-Batch*, and *VS-Batch+* are `8700.6`, `11349.0`, `86.2`, `8697.4`, `8357.2`, and `7255.0`, respectively. In summary, *VS-Batch+* reduces the number of cache thrashing events by more than `13.2%`, in contrast to other comparison schemes. This is because *VS-Batch+* keeps the neighboring data pages of currently accessed data pages in the SSD cache with batches, with an adaptive

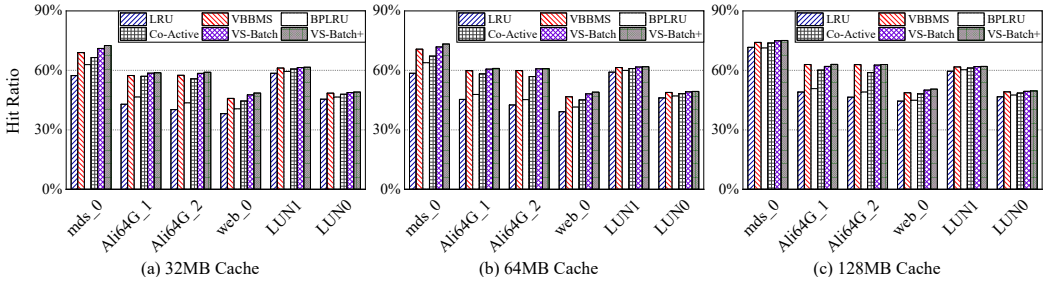


Fig. 10. Comparison of cache hit ratio with varied cache management polices.

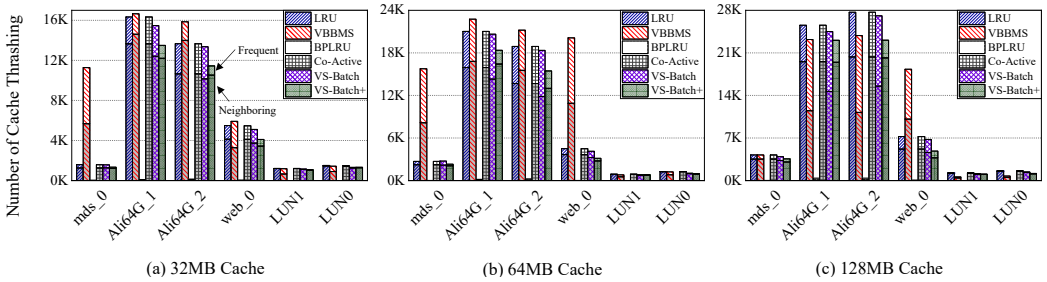


Fig. 11. Comparison of cache thrashing with varied cache management polices.

reconstruction of visibility graph of cached pages. In summary, our strategy leverages the spatial locality of reference, which reduces cache thrashing and thus improves the cache hits ratio.

There are two noticeable clues shown in Figure 11. The first is that *VBBMS* greatly increase cache thrashings comparing to other schemes, after replaying the block traces. This is because *VBBMS* manages the cached data pages as virtual blocks, and each virtual block unconditionally has more than 6 pages. Thus, it worsens cache thrashing while increasing cache hits for sequential accesses.

Another clue is about, *BPLRU* produces the least number of cache thrashing after replaying the traces. This is because it manages the cached data in the granularity of block, which contains 256 pages in our tests, and then the data pages are more likely to be ejected together with the nearby data pages, rarely resulting in cache thrashing.

4.3.2 I/O Latency. Figure 12 demonstrates the results of overall I/O time, and the schemes of *LRU*, *VBBMS*, *BPLRU*, *Co-Active*, *VS-Batch*, and *VS-Batch+* lead to the overall I/O latency of `4.2ks`, `3.2ks`, `3.9ks`, `3.3ks`, `3.0ks`, and `2.6ks`, respectively. Clearly, the proposed *VS-Batch+* cache management approach outperforms others regarding the measure of the overall I/O time. More precisely, *VS-Batch+* can cut down the overall I/O latency including read latency and write latency, by `38.6%`, `19.4%`, `33.9%`, `22.9%`, and `13.0%` on average, compared to *LRU*, *VBBMS*, *BPLRU*, *Co-Active*, and *VS-Batch*.

It is worth mentioning that all cache schemes cause high average I/O latencies for replaying the MSRC traces of *mds_0* and *web_0*, by comparing to running other traces. We suggest that both *mds_0* and *web_0* have very small values of the interval between two requests in a major part of workloads (refer to Table 3), implying congested I/O workloads in them, so that they requires a large average I/O latency for servicing a read/write request.

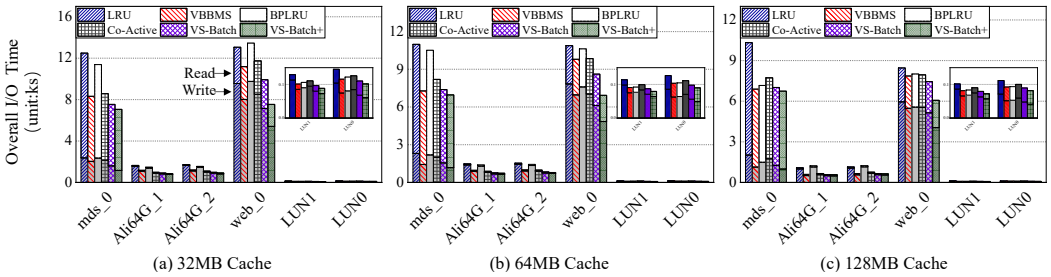


Fig. 12. Comparison of overall I/O response time.

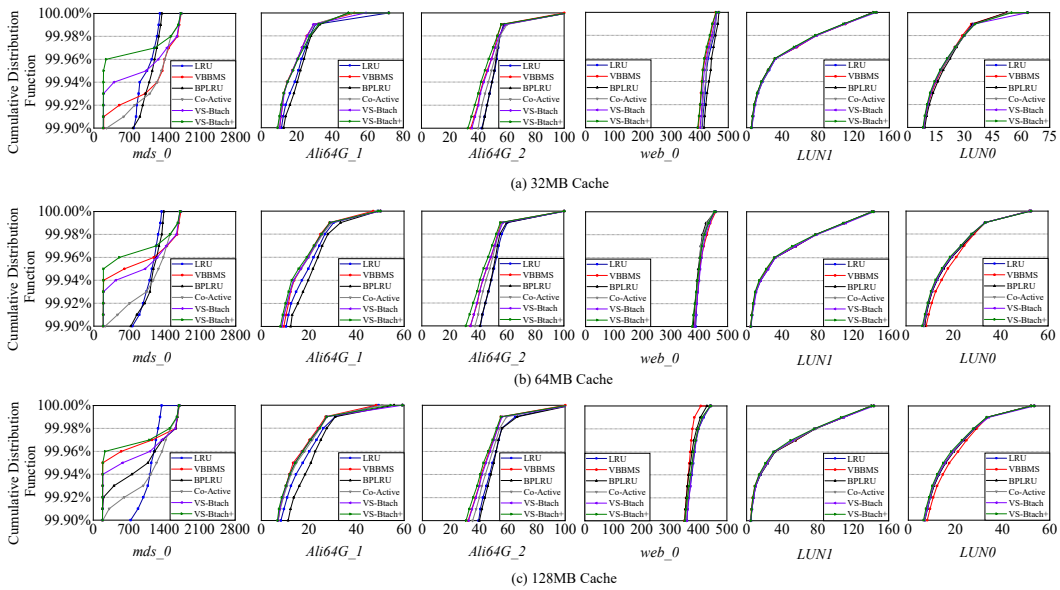


Fig. 13. Cumulative distribution function of I/O response time. The unit on the X-axis of all sub-figures is millisecond.

The measure of long-tail latency is another critical indicator of SSD I/O performance, which is commonly expressed in the form of a Cumulative Distribution Function (*CDF*). We collected the results of long-tail latency after replaying the selected traces, and Figure 13 reports the CDF of the slowest `0.1`% of the I/O requests. As read, *VS-Batch+* can improve the tail latency compared to other scheme by between `1.0`% and `2.1`%, since it can slightly mitigate the negative impact of I/O requests caused by bursty requests, by absorbing some of them in the cache. Thus, the tail latency can be consequently improved. We argue that the slowest 0.1% requests of benchmarks are mainly caused by I/O congestion, and the cache management approach may not be able to substantially relieve the congestion level for such worst cases.

4.3.3 GC Statistics and Analysis. We record the number of GC operations after running all selected traces with varied cache polices, and Figure 14 reports the results. As seen, *VS-Batch+* does not lead to more GC operations, compared with *LRU*, *Co-Active*, and *VS-Batch*. This fact verifies visibility graph-based batch eviction can group the cached data

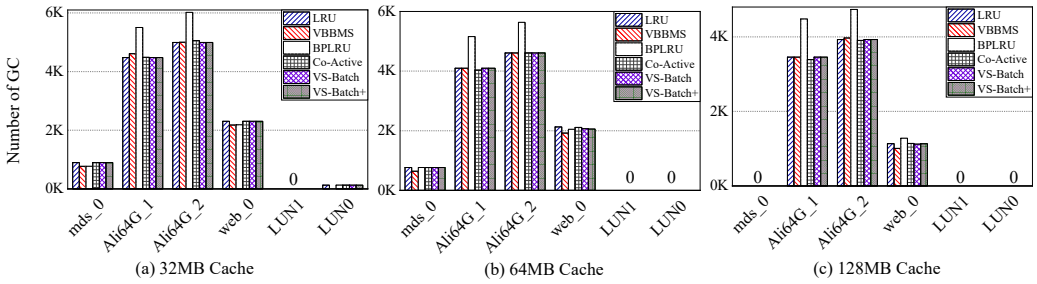


Fig. 14. Comparison of GC statistics after running the selected traces with varied cache polices.

pages that have a low probability of re-accessing in the future. Consequently, *VS-Batch+* does not result in more flush operations to the underlying flash array of SSDs, causing not more GC operations in the end.

In general, batch eviction of *VS-Batch+* does not noticeably perform worse than *VBBMS* on the measure of GC count by no more than `5.2%` on average, after running all benchmarks. Another interesting clue is that, *VBBMS* bring about an obvious reduction of GC operations in some specific runs of benchmarks, though they cause more cache thrashing events, when comparing to our proposal. For example, in the case of *web_0*, our approach brings about more GC operations by `10.0%` on average, in all cache size configurations, when comparing to the most related work of *VBBMS*. This is because both related cache management schemes attach more importance to temporal reference of write accesses, and thus achieve better write hits that has been previously illustrated in Section 4.3.2. As a result, the related work of *VBBMS* yields less GC operations after replaying some benchmarks, in contrast to our proposal of *VS-Batch+*. We argue that *BPLRU* manages the cached data as the granularity of virtual block, that consists of *256* cached data pages, so that it has a greater chance of keeping infrequently accessed pages in cache, and then triggers more erase operations.

4.3.4 Space and Time Overhead. The space overhead depends on the size of SSD cache and the processing granularity, all cache management schemes expect the same size of memory for managing the cached items whether using one linked list or multiple lists. Both *VS-Batch+* and *VS-Batch*, however, need to additionally store the matrix of visibility graph and the four linked lists. The visibility matrix consumes at most `128KB` (= `4096` (nodes) * (`64+64`) (visible nodes) * `2bit` / `8`) in the case of *32MB* cache. And linked lists contain two links (`2*4B`) and page addresses (`8B`), which need `64KB` in the case of `32MB` cache. Besides, *VS-Batch* has to consume `1Bit` per page to record access status.

Table 4 summarizes the space overhead of all cache management methods with varied cache configurations. As seen, all used cache management methods have a similar space overhead, mainly caused by saving metadata of cached data pages. A noticeable clue is that, *BPLRU* results in the least space overhead, this is because it deals with the cached data pages in the unit of block, which contributes to the reduction of the length of linked list for managing the cached data pages.

Considering *LRU*, *VBBMS*, *BPLRU*, and *Co-Active* do not require to generate visibility graph of cached data pages, we only record the time overhead of *VS-Batch+* and *VS-Batch* with various cache configurations, and report the results in Figure 15. As read, both schemes cause an average of no more than `5.7` μs per I/O request, or less than `2.0%` of the overall I/O time. More exactly, comparing with *VS-Batch*, *VS-Batch+* generally results in more time overhead when running the write-intensive workloads with cache configurations of `32MB` and `64MB`, since it triggers more

Table 4. Space Overhead (unit:KB)

Cache Policies	32MB	64MB	128MB
LRU	224	448	896
VBBMS	62	124	248
BPLRU	0.9	1.8	3.5
Co-Active	228	456	912
VS-Batch	356	712	1424
VS-Batch+	356	712	1424

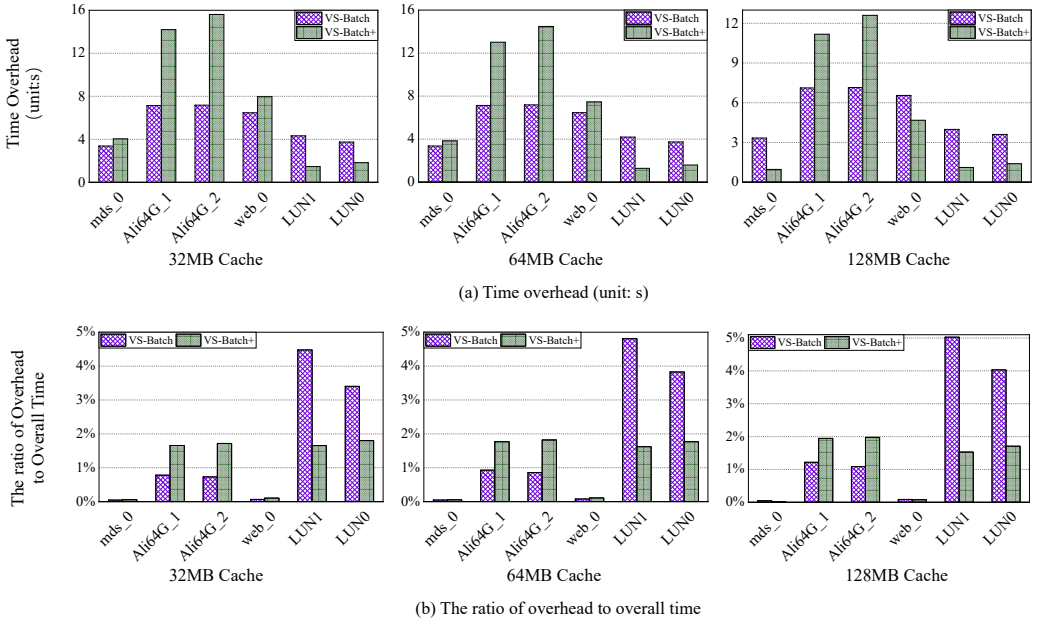


Fig. 15. Extra time cost of different cache management schemes. Note that *LRU*, *VBBMS*, *BLPRU*, and *Co-Active* do not incur extra computations except for traversing LRU lists.

reconstructions of visibility graph. But, *VS-Batch+* can save time overhead by up to 71.0% while the cache size becomes 128MB, as the number of visibility graph reconstructions is less than that of *VS-Batch*.

In brief, we argue that the time overhead caused by our proposal is acceptable, even though it runs on a compute power-limited platform. Note that the computation overhead does bring about impacts on I/O response time by postponing dispatch on incoming I/O requests, and relevant I/O time results have been previously reported in Section 4.3.2.

5 RELATED WORK

Many caching policies are continuously being designed to suit the different needs of applications, this section depicts cache management for conventional storage and flash memory, separately.

5.1 Advanced cache schemes in conventional storage

A number of advanced cache management policies have been proposed for conventional storage systems, such as multi-queue replacement *(MQ)* [38], *ARC* [14], *DULO* [17], wise ordering for writes *(WOW)* [39], low interference recency set (*LIRS*), and power-aware cache management schemes [41]. Some of them are adaptive, such as *MQ* and *ARC*, while others try to make optimizations for specific workloads or specific application scenarios. For example, *DULO* is an effective cache management scheme to exploit both temporal and spatial locality of workloads, by considering sequential access is more efficient than random access. To this end, *DULO* gives high priority to random blocks by preferentially evicting sequential blocks that have similar logical addresses and access time.

5.2 Emerging cache schemes in flash memory

With the advancement in flash memory, certain specific cache management approaches have been introduced to gain better overall performance of SSDs. Typical flash-aware cache management approaches incorporate the structure or state of underlying flash arrays and aim to reduce the I/O latency of SSDs. We summarize them into two categories by considering the management granularity is a page (fine-grained) or multi-pages (coarse-grained).

Fine-grained cache schemes in flash memory. According to the asymmetrical eviction cost of clean and dirty pages, *CFLRU* [11] 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 cost. Similar to [42], Chen et al. [43] presented *ECR*, that gives a higher probability to evict a page when it needs the shortest waiting time in the corresponding chip (or channel) queue. Besides, Wang et al. [5] introduced a scheme for the management of SSD cache with consideration of the access frequency of the buffered data pages. They used the particle swarm optimization (PSO) technique [13] to predict the access frequency of the buffered pages for guiding cache evictions.

Sun et al. [7] proposed *Co-Active*, which considers the I/O access patterns of applications and the idle status of flash chips, to determine which buffered data pages should be evicted. Specifically, it proactively evicts the (cold) cached items from the cache if their destination (underlying) SSD channels are idle (i.e. the spatial factor), for reducing the wait time of flushing data. Thus, it can minimize the delay on normal I/O processing caused by cache evictions and then reduce I/O latency.

Coarse-grained cache schemes in flash memory. Du et al. [16] proposed a cache management scheme called *VBBMS* that manages the cached data pages at the granularity of virtual blocks, and the data pages that reside in the same virtual block may have a similar access pattern. Moreover, it divides the cache into two regions for responding random requests and sequential requests, and employs *LRU* and first in first out (*FIFO*) for selecting the victims of virtual blocks to be evicted in two regions of cache. Furthermore, considering that the garbage collection (GC) process of NAND-based SSDs is performed at the level of block that consists of multiple SSD pages, SSD block-level cache management approaches including Flash-Aware Buffer (*FAB*) [44], *PUD-LRU* [45], and block padding least recently used (*BPLRU*) [37] have been proposed to better exploit spatial locality.

Besides, Shim et al. [46] proposed an adaptive partitioning scheme, which is based on a ghost caching mechanism, to adaptively tune the ratio of the buffering and the mapping space in the device cache according to the workload characteristics.

6 CONCLUSIONS

This paper proposes a visibility graph-based cache management scheme for SSDs, called *VS-Batch+*. It unifies both temporal and spatial locality of references, and supports batch adjustment of adjacent or nearby hot cached data by referring to connection situations in the visibility graph of all cached data pages. Specially, it can adaptively refresh the visibility graph of cached data pages according to the ratio of evicted data pages since the last round of visibility graph reconstruction, to direct cache management in a timely manner. Besides, *VS-Batch+* enables evicting the cold access data from the cache in the unit of batch by also resorting to the visibility graph, to maximize the internal flushing parallelism of SSD devices.

Through a series of simulation tests based on several real-world disk traces, we show that our proposal can noticeably enhance the cache hits by `4.2%` on average, and then reduce I/O latency by between `13.0%` and `38.6%`, compared with the state-of-art cache management schemes for SSDs. In the future, we will explore more rules on the basis of visibility graph by taking the features of I/O workloads into account, to better support batch adjustment of cached pages for the DRAM buffer inside SSDs.

ACKNOWLEDGMENTS

This work was partially supported by “National Natural Science Foundation of China (No. 61872299) and “the Natural Science Foundation Project of CQ CSTC (No. cstc2021ycjh-bgzxm0199, 2022NSCQ-MSX0789)".

REFERENCES

[1] Kim B., Choi J., and Min S. Design tradeoffs for SSD reliability. In *FAST*, 2019.
[2] Xu X., Cai Z., Liao J., and Ishiakwa Y. Frequent access pattern-based prefetching inside of solid-state drives. In *DATE*, 2020.
[3] Kim K., and Kim T. HMB in DRAM-less NVMe SSDs: Their usage and effects on performance. In *PloS one*, 2020.
[4] Tarihi M., Asadi H., and Haghdoost A. et al. A hybrid non-volatile cache design for solid-state drives using comprehensive I/O characterization. In *IEEE TC*, 2016.
[5] Wang Y., Kim K., and Lee B. et al. A novel buffer management scheme based on particle swarm optimization for SSD. In *TJSC*, 2018.
[6] Li J., Sha Z. and Cai Z. et al. Patch-Based Data Management for Dual-Copy Buffers in RAID-Enabled SSDs. In *IEEE TCAD*, 2020.
[7] Sun H., and Dai S. et al. Co-Active: A Workload-Aware Collaborative Cache Management Scheme for NVMe SSDs. In *IEEE TPDS*, 2021.
[8] Jain A., and Lin C. Back to the future: Leveraging Belady’s algorithm for improved cache replacement. In *ISCA*, 2016.
[9] Kandemir M., and Ramanujam J. et al. Improving cache locality by a combination of loop and data transformations. In IEEE *TC*, 1999.
[10] Wu G., He X., and Eckart B. An adaptive write buffer management scheme for flash-based ssds. In *ACM TOS*, 2012.
[11] Park S., Jung D., and Kang J. et al. CFLRU: a replacement algorithm for flash memory. In *CASES*, 2006.
[12] Robinson J., and Devarakonda M. Data cache management using frequency-based replacement. In *SIGMETRICS*, 1990.
[13] Khan S. U., Yang S., and Wang L. et al. A modified particle swarm optimization algorithm for global optimizations of inverse problems. In *IEEE TOM*, 2015.
[14] Megiddo and Modha D. A self-tuning low overhead replacement cache. In *FAST*, 2003.
[15] Wang M., and Li Z. A spatial and temporal locality-aware adaptive cache design with network optimization for tiled many-core architectures. In *IEEE VLSI*, 2017.
[16] Du C., Yao Y., Zhou J., and Xu X. VBBMS: A novel buffer management strategy for NAND flash storage devices. In *IEEE TCE*, 2019.
[17] Jiang S., Ding X,. and Chen F. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Locality. In *FAST*, 2005.
[18] Wang H., Yi X., and Huang P. et al. Efficient SSD caching by avoiding unnecessary writes using machine learning. In *ICPP*, 2018.
[19] Lacasa L., Luque B., and Ballesteros F. et al. From time series to complex networks: The visibility graph. In *PNAS*, 2008.
[20] Sha Z., Cai Z., and Yin D. et al. Unifying Temporal and Spatial Locality for Cache Management inside SSDs. In *DATE*, 2022.
[21] Tran N, Reed D, Member S. Automatic ARIMA time series modeling for adaptive I/O prefetching. In *TPDS*, 2004.
[22] Gao, Y., Yu, D., and Wang, H. Fault diagnosis of rolling bearings using weighted horizontal visibility graph and graph Fourier transform. In *Measurement*, 2020.
[23] Iacobello, G., Scarsoglio, S., and Ridolfi, L. Visibility graph analysis of wall turbulence time-series. In *Physics Letters A*, 2018.
[24] Donner, R. V., and Donges, J. F. Visibility graph analysis of geophysical time series: Potentials and possible pitfalls. In *Acta Geophysica*, 2012.
[25] Lacasa L., Luque B., and Ballesteros F. et al. Horizontal visibility graphs: Exact results for random time series. Phys. Rev., 2009.

[26] Li, H., et al. Merging and Prioritizing Optimization in Block I/O Scheduling of Disk Storage. In *JCSC*, 2021.
[27] Liao, J., Trahay, F., Gerofi, B., and Ishikawa, Y. Prefetching on storage servers through mining access patterns on blocks. In *TPDS*, 2016.
[28] Erdos P., Faudree R., Pach J., and Spencer J. How to make a graph bipartite. In *J. Combin. Theory Ser*, 1988.
[29] Zhang W., Cao Q., and Jiang H. et al. PA-SSD: A Page-Type Aware TLC SSD for Improved Write/Read Performance and Storage Efficiency. In *ICS*, pp. 22-32, 2018.
[30] Liu C., and Lee Y.,et al. GSSA: A Resource Allocation Scheme Customized for 3D NAND SSDs. In *HPCA*, 2021.
[31] Chang, T., Hsieh, J. W., and Chang, T. C., et al. EMT: Elegantly Measured Tanner for Key-Value Store on SSD. In *TCAD*, 2021.
[32] Wang W., Chen T., and Chang Y H., et al. How to cut out expired data with nearly zero overhead for solid-state drives. In *DAC*, 2020.
[33] Narayanan D., Donnelly A., and Rowstron A. Write off-loading: Practical power management for enterprise storage. In *ACM TOS*, 2008.
[34] Lee C., and Matsuki T. et al. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *ACM SYSTOR*, 2017.
[35] Alibaba Block Traces. https://github.com/alibaba/block-traces.
[36] Kim B., Yang H., and Min S. AutoSSD: an Autonomic SSD Architecture. In *ATC*, 2018.
[37] Kim H. and Ahn S. BPLRU: A buffer management scheme for improving random writes in flash storage. In *FAST*, 2008.
[38] Zhou Y., and Philbin J. The multi-queue replacement algorithm for second level buffer caches. In Proceedings of the USENIX Annual Technical Conference, pp. 91–104, 2001.
[39] Gill B. and Modha D. WOW: Wise ordering for writes-combining spatial and temporal locality in non-volatile caches. In Proceedings of the USENIX Conference on File and Storage Technologies (*FAST*), 2005.
[40] Jiang S., and Zhang X. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. ACM SIGMETRICS Performance Evaluation Review, Vol. 30(1), 31-42, 2002.
[41] Zhu Q., David F., and Devaraj C. et al. Reducing energy consumption of disk storage using power-aware cache management. In 10th international symposium on high performance computer architecture (*HPCA*), pp. 118-118, 2004.
[42] Wu S., and Mao B. et al. Garbage collection aware cache management with improved performance for flash-based SSDs. In *ICS*, 2016.
[43] Chen H., Pan Y., and Li C. et al. ECR: Eviction-cost-aware cache management policy for page-level flash-based SSDs. In *CCPE*, 2019.
[44] Jo H. et al. FAB: Flash-aware buffer management policy for portable media players. In *TCE*, 2006.
[45] Hu J. et al. PUD-LRU: An erase-efficient write buffer management algorithm for flash memory SSD. In *MASCOTS*, 2010.
[46] Shim H., Seo B., and Kim J. et al. An adaptive partitioning scheme for DRAM-based cache in solid state drives. In *MSST*, 2010.