Unifying temporal and spatial locality for cache management inside SSDs

Zhibing Sha, Zhigang Cai, François Trahay, Jianwei Liao, Dong Yin

Unifying Temporal and Spatial Locality for Cache Management inside SSDs

Zhibing Sha*, Zhigang Cai*, François Trahay†, Jianwei Liao*, Dong Yin‡
*College of Computer and Information Science, Southwest University, Chongqing, China
†Telecom SudParis, Institut Polytechnique de Paris, France
‡School of Computer Science and Technology, Huaihua University, Huaihua, China
Corresponding author: Z. Cai (*czg@swu.edu.cn*); D. Yin (*yindong2050@126.com*)

***Abstract*—To ensure better I/O performance of solid-state drivers (SSDs), a dynamic random access memory (DRAM) is commonly equipped as a cache to absorb overwrites or writes, instead of directly flushing them onto underlying SSD cells. This paper focuses on the management of the small amount cache inside SSDs. First, we propose to unify both factors of temporal and spatial locality of user applications by employing the visibility graph technique, for directing cache management. Next, we propose to support batch adjustment of adjacent or nearby (hot) cached data pages by referring to the connection situations in the visibility graph of all cached pages. At last, we propose to evict the buffered data pages in batches, 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 more than `2.8%`, and the overall I/O latency by `20.2%` on average, in contrast to conventional cache schemes inside SSDs.**

***Index Terms*—Solid-state Drivers, Cache Management, Locality of Reference, Visibility Graph, Batch Adjustment.**

I. Introduction

The NAND flash memory-based solid-state drives (SSDs) have gradually become the dominant storage devices for embedded systems, personal computers, and high performance platforms, thanks to their small size, high performance, random-access and low energy consumption [1], [2]. Apart from NAND flash arrays that hold 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`$\sim$`480MB` cache [3].

Generally, the SSD cache is utilized to not only keep logic-physical address mapping data structures, but also temporarily buffer the contents of overwrite or write requests [4]. The write requests can be quickly responded after their contents are buffered in the cache, Consequently, it can greatly reduce the number of flush operations onto the underlying flash array, and then improve the I/O performance and the lifetime of SSDs [5], [6]. Once the cache space is full, the cache management scheme must evict some buffered data and flush them to the underlying flash array, for making room for new data.

Cache management significantly impacts the I/O performance because of the limited cache capacity in SSDs [7]. Locality of reference characterizes 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 [8].

Least recently used (*LRU*) is the most widely used cache management scheme due to the simplicity and adaptability [9]. It is based on 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 [10]. Similarly, the *LFU* cache management 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 [11].

With respect to sophisticated cache management in SSDs, Sun et al. [6] 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. [4] 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 [14] to predict the access frequency of pages, in order to guide cache evictions. In addition, Du et al. [15] proposed a virtual block-based buffer management scheme for SSD devices, that groups the buffered data pages into virtual blocks according to their access patterns (i.e. random or sequential), and manages the pages at the virtual block level.

Although such sophisticated methods can improve the cache use efficiency in many cases compared to *LRU* or *LFU*, their computational power consumption is increased because of the analysis of I/O workloads or the monitoring the idle status of SSD devices. As a result, they cannot cover all scenarios with expected I/O improvements [17]. Considering SSD devices normally have computing power-limited controllers, it is expected to integrate a simple and effective cache management scheme with such devices.

On the other hand, most workloads have a high spatial locality and temporal locality, and designing a cache that leverages this locality can boost the storage performance in computing systems [16]. The spatial locality of reference, however, has not been utilized together with the factor of temporal locality in cache management for SSDs.

In this paper, we propose *VS-Batch*, a cache management scheme for SSDs devices that considers both temporal and spatial locality of references. *VS-Batch* unifies both types of locality by using a visibility graph. In summary, it makes the following three contributions:

- We propose to use the ***visibility graph*** technique [18] 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 upgrade and downgrade adjustment on the cached data pages. It upgrades the hit pages, their neighboring pages and their (nearby) frequently accessed pages to high-level linked lists ***in batches***, and it degrades the cold data pages to low-level linked lists till they are evicted from the cache.
- We conduct simulation evaluation by replaying `6` block traces of real world applications. As our measurements indicate, *VS-Batch* effectively increases cache hits by more than `2.8%` and reduces the I/O latency by more than `10.7%`.

The remainder of the paper is organized as follows: in Section II, we describe the related work and motivation. Section III presents the proposed cache management scheme that takes both temporal and spatial locality into account. Section IV presents the evaluation methodology and reports the experimental results. Section V concludes the paper.

II. Related Work and Motivation

A. Related Work

Caching inside of 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 cache replacement strategies are basically built on the top of temporal locality, such as *LRU*, *CFLRU*, or *LFU*. Moreover, Sun et al. [6] proposed considering I/O access patterns of applications and the idle status of flash chips, to determine which buffered data pages should be evicted. Then, 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. Similar to [12], Chen et al. [13] 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. [4] introduced a scheme for the management of SSD cache with consideration of the access frequency of the buffer pages. They used the particle swarm optimization (PSO) technique [14] to predict the access frequency of the buffered data pages for guiding cache evictions.

Du et al. [15] proposed a buffer management scheme called *VBBMS* that takes both temporal and spatial factors into account. It 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.

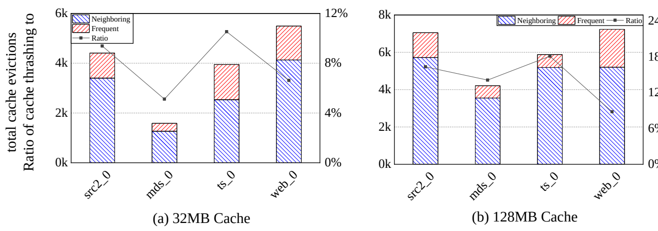


Fig. 1: The number of cache thrashings and the proportion of thrashing events to the total evictions (with *LRU*). The labels of *Neighboring* and *Frequent* correspondingly represent the thrashed items are either neighbors and nearby frequently accessed data pages.

The sophisticated cache management methods cost computing power for analyzing I/O workloads or identifying hot/cold data, to make an "accurate" eviction decision. We emphasize that such approaches are not good solutions for computing power-limited SSD devices, and it is necessary to propose a simple and effective cache management scheme for SSD devices, by unifying both temporal and spatial localities.

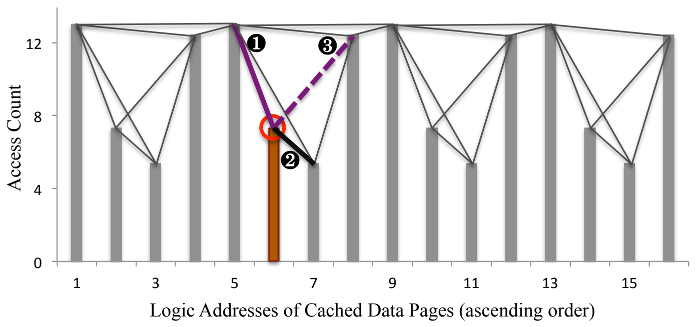
B. Motivations

As discussed, existing cache management approaches for SSDs, only take the temporal locality of reference (or with the spatial factor) into account when carrying out cache replacement[1]. 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. Read Section IV-A about the specifications on the experimental platform and benchmarks.

We first define a term of ***cache thrashing***, and a cache thrashing event indicates ***while*** a specific data page is invariably kept in the SSD cache, 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 traces, we record the number of cache thrashing events and count the ratio of such events to the total evicted data items. Figure 1 shows the results when using the commonly used *LRU* scheme.

In this experiment, it brings about `1585`$\sim$`7233` cache thrashing events, taking up to `18.1%` of total evictions, after running the selected traces. It is possible to avoid a cache thrashing case if the thrashed data item is managed in a batch with its in-cached neighboring or nearby data items. Thus, we summarize that the spatial locality of cached pages is worth

[1]Though *VBBMS* [15] declares 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 locality of references.



(a) Visibility graph

$$\begin{array}{c} \\ \textcircled{1} \\ \cdots \\ \textcircled{5} \\ \textcircled{6} \\ \textcircled{7} \\ \textcircled{8} \\ \cdots \end{array}\begin{array}{c} \begin{array}{cccccccc} \textcircled{1} & \ldots & \textcircled{5} & \textcircled{6} & \textcircled{7} & \textcircled{8} & \textcircled{9} & \ldots \end{array} \\ \left[\begin{array}{cccccccc} 00 & \ldots & 11 & 00 & 00 & 00 & 00 & \cdots \\ \cdots & & & & & & & \\ \cdots & & 00 & 01 & 00 & 01 & 11 & \cdots \\ \cdots & & \mathbf{01} & \mathbf{00} & \mathbf{10} & \mathbf{11} & \mathbf{00} & \cdots \\ \cdots & & 11 & 01 & 00 & 01 & 00 & \cdots \\ \cdots & & 11 & 00 & 10 & 00 & 01 & \ldots \\ \cdots & & & & & & & \end{array}\right] \end{array}$$

Two−bit label Indication

11: not adjacent & higher freq.
10: adjacent & lower freq.
01: adjacent & higher freq.
00: others

(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.*

considering in cache management of SSDs, to boost cache use efficiency and I/O performance.

Such observation motivates us to build an efficient and simple approach on the top of temporal locality-based cache management for SSDs, by also unifying the spatial locality of reference. 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.

III. Visibility Graph-based Cache Management

A. 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.

To this end, we first leverage visibility graph [18] 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. 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.

B. Design and Implementation of VS-Batch

1) Visibility Graph: In order to model the temporal and spatial localities of all cached data pages, we employ the visibility graph technique, which was proposed to generate mapping networks from time series [18]. 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_b, y_b) will have visibility, and then become two connected nodes in the graph, if any other data (x_c, y_c) placed between them fulfills:

$$y_c < y_b + (y_a - y_b) \times \frac{x_b - x_c}{x_b - x_a} \tag{1}$$

where x_i indicate 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: (1) adjacent (the nearest neighbors) and higher access frequency (eg. node #5), (2) adjacent and lower access frequency (eg. node #7), and (3) not adjacent and higher access frequency (eg. node #8). Then, we adjust the "visible" data pages of a data page being accessed, according to its visibility type. The visibility graph is stored as a matrix that depicts the connection between couples of data pages, as shown in Figure 2(b).

2) Batch Adjustment: 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 ascending order. Basically, 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 head of *Eviction list*. When a data page is accessed, *VS-Batch* moves the corresponding node to the tail of the *Hit list*, and adjusts the nodes that can be seen in the visibility graph. To be specific, the visible and high frequency access nodes will be moved to the *Hot list*, and other adjacent nodes are moved to the *Adjacent list*.

At the initialization stage (i.e. all lists are empty), all the nodes of cached data pages are linked in the *Eviction list* with a *LRU* fashion. Once a given cached page is hit again, *VS-Batch* carries out the upgraded batch adjustment, to move the nodes of hit page and their visible (cached) pages into higher-level lists. As the example in Figure 3(a) shows, once **Page *F*** is hit, its node is directly moved to tail of the *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

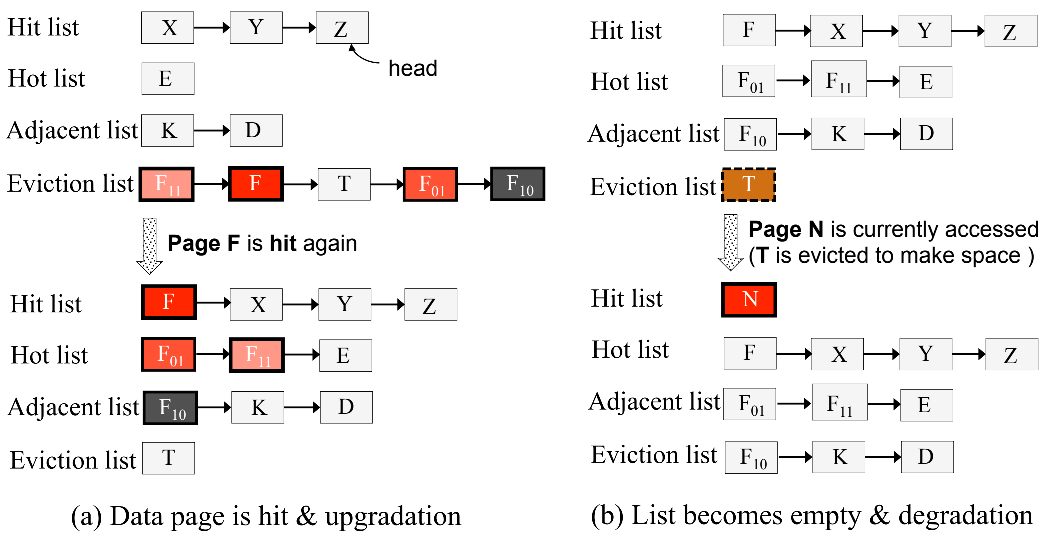


Fig. 3: Batch adjustment of cached items in *VS-Batch* (assuming the SSD cache buffers only `11` data pages in total for illustration simplicity). In the node, the identifier with a subscript indicates the connection case in the visibility graph, from the view of the identifier page.

Algorithm 1: Batch Adjustment in *VS-Batch*

```
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 src_level $\leq$ dst_level then
11             /*move to the head of destination list*/
               move_to_list(addr, dst_level);
12         end
13     end
```

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 count.

On the other side, *VS-Batch* also conducts hierarchical downgraded movements when a lower-level list becomes empty. As illustrated in Figure 3(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 the head of the *Eviction list* (i.e. **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 *Hot list*.

3) Implementation: Algorithm 1 illustrates the details on batch adjustment on the nodes in four-level linked lists. As shown, *Lines 1-13* 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.

IV. Experiments and Evaluation

A. Experiment Setup

We have performed trace-driven simulation with *SSDsim (ver*2.1*)* [19], 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 I shows our settings of experiments, by mainly referring to [2], [15]. To further investigate how our proposal works with varied scales of SSD cache, we set the cache size as `32MB` and `128MB`.

To evaluate *VS-Batch*, we employ `6` commonly used disk traces. `4` of these traces are from the block I/O trace collection of Microsoft Research Cambridge [20]. The remainder two block I/O traces are recently collected from a part of an enterprise virtual desktop infrastructure (VDI) [21]. Specifically, they are additional-01-1619-LUN0 (*LUN0*) and additional-01-1620-LUN0 (*LUN1*). The detailed specifications on the traces are shown in Table II.

Apart from *LRU* and the proposed *VS-Batch* approach, the following two schemes are also used in comparison evaluation:

- *VBBMS* [15] 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. We argue that *VBBMS* is the most related work of our approach.
- *Co-active* [6] first classifies the data into hot and cold categories, by referring to the factor of temporal locality.

TABLE I: Experimental settings of *SSDsim*

Parameters	Values	Parameters	Values
Channel Size	8	*Read latency*	0.075ms
Chip Size	4	*Write latency*	2ms
Plane Size	4	*Erase latency*	15ms
Block per plane	256	*Transfer (Byte)*	10ns
Page per block	256	*GC Threshold*	10%
Page Size	8KB	*DRAM Cache*	32/128
FTL Scheme	Page level		

TABLE II: Specifications on traces (ordered by write ratio)

Traces	Req #	Wr Ratio	Wr Size	Frequent R (Wr)
src2_0	1557814	88.6%	7.1KB	32.6%(58.5%)
mds_0	1211034	88.1%	7.2KB	0.3%(49.8%)
ts_0	1801734	82.4%	8.0KB	28.5%(41.4%)
web_0	2029945	70.1%	8.6KB	22.7%(71.3%)
LUN1	1124327	41.6%	12.3KB	2.9%(6.9%)
LUN0	1430675	41.1%	11.6KB	6.9%(4.7%)

Note: **Frequent R** means the ratio of addresses requested not less than 3, and **(Wr)** implies the percent of write addresses in which.

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.

For periodically generating the visibility graph of buffered data pages, we perform a round of visibility graph processing when the total amount of write data of I/O requests becomes greater than the size of cache, since the last processing round. Then, we employ the obtained visibility graph to direct cache management in the forthcoming time window. Considering the range of spatial locality and the overhead of visibility graph processing, we only 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.

B. Results and Discussion

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. In other words, the write request can be completed with a lower latency if its contents can be directly absorbed in the cache.

Figure 4 reports the cache hits ratio after running the benchmarks with varied cache management schemes. As illustrated, *VS-Batch* performs the bests, and achieves an improvement on cache hits by 15.9%, 2.8%, and 5.1%, compared to *LRU*, *VBBMS*, and *Co-active*. To further analyze why *VS-Batch* improves the cache hit ratio, Figure 5 reports the number of cache thrashing events. On average, *VS-Batch* reduces the number of cache thrashing events by 23.1%, 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. In summary, our strategy leverages the spatial locality of reference, which reduces cache thrashing and thus improves the cache hits ratio.

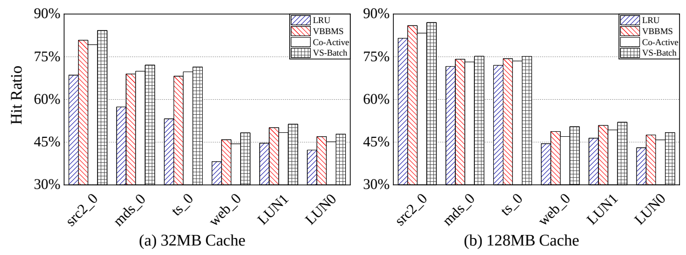


Fig. 4: Comparison of cache hit ratio with varied cache configurations (note that Y-axis starts from *30%*).

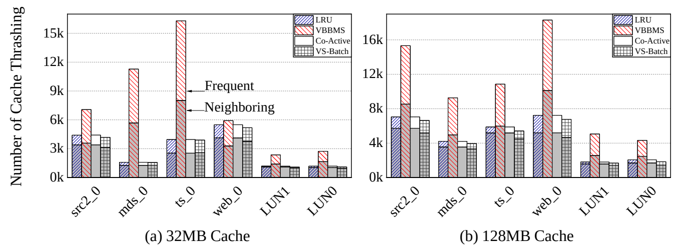


Fig. 5: Comparison of cache thrashing.

Another noticeable clue shown in Figures 4 and 5 is about that *VBBMS* causes the largest number of cache thrashings but not the worst hit ratio. This is because *VBBMS* manages the cached data pages as virtual blocks, and each virtual block unconditionally has more than 6 pages. So it worsens cache thrashing while increasing cache hits for sequential accesses.

2) I/O Latency: Figure 6 presents the results of overall I/O time. 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 by 27.9%, 10.7%, and 22.1% on average, in contrast to *LRU*, *VBBMS*, and *Co-active*. Additionally, the performance improvement of *VS-Batch* increases when the cache size becomes large, which demonstrates its scalability.

For read-intensive workloads of *LUN0* and *LUN1*, the performance of *VS-Batch* and other comparison counterparts are similar. We argue that both trace have around 41.3% of write requests, that limits the room for I/O improvements with the cache. On the other hand, *VS-Batch* can bring about more than 11.8% reduction of I/O response time for the selected write-intensive workloads. This confirms that *VS-Batch* offers better cache use efficiency (i.e. more cache hits), which contribute to the reduction in I/O latency for write-heavy workloads.

3) Overhead: The main memory overhead of *VS-Batch* is due to the storage of the matrix of visibility graph and the four linked lists. This overhead depends on the size of SSD cache. The visibility matrix consumes at most 128KB (= 4096(nodes) * (64+64) (visible nodes) * 2bit / 8) in the case of *32MB* cache. Besides, linked lists contain two links (2*4B) and page addresses (8B), which need 64KB in the case of *32MB* cache. Note that all cache management schemes expect the same size of memory for managing the cached items, whether using one linked list or multiple lists.

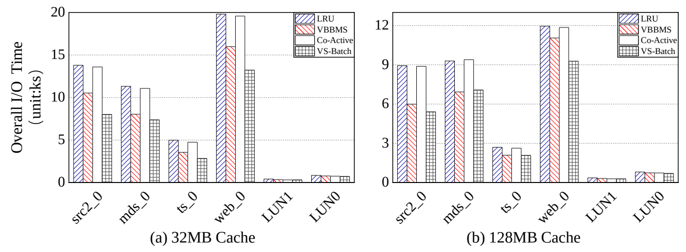


Fig. 6: Comparison of overall I/O response time.

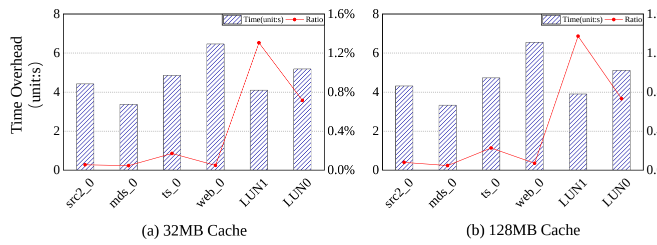


Fig. 7: Time overhead of batch adjustment in *VS-Batch*.

We record the time overhead of *VS-Batch* with various cache configurations, and report the results in Figure 7. As read, it causes an average of 3.1 μs per I/O request, or less than 1.3% of the overall I/O time. This is because *VS-Batch* brings about more operations on the linked lists due to batch adjustment in cache management, as well as reconstructions of visibility graph.

Thus, we conclude 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 IV-B2.

V. 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. Then, it can noticeably cut down the number of cache thrashing and thus reduce the I/O latency, when running user applications.

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 more than 2.8%, and then reduce I/O latency by between 10.7% and 27.9%, compared with the state-of-art cache management schemes for SSDs.

Acknowledgment

This work was partially supported by "National Natural Science Foundation of China (No. 61872299, No. 62032019)", "Chongqing Talents: Exceptional Young Talents Project (No. CQYC202005094)", "the Natural Science Foundation Project of CQ CSTC (No. cstc2021ycjh-bgzxm0199)", and "Chongqing Graduate Research and Innovation Project (No. CYS20117)".

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] Wang Y., Kim K., and Lee B. et al. A novel buffer management scheme based on particle swarm optimization for SSD. In *TJSC*, 2018.
[5] 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.
[6] Sun H., and Dai S. et al. Co-Active: A Workload-Aware Collaborative Cache Management Scheme for NVMe SSDs. In *IEEE TPDS*, 2021.
[7] Jain A., and Lin C. Back to the future: Leveraging Belady's algorithm for improved cache replacement. In *ISCA*, 2016.
[8] Kandemir M., and Ramanujam J. et al. Improving cache locality by a combination of loop and data transformations. In IEEE *TC*, 1999.
[9] Wu G., He X., and Eckart B. An adaptive write buffer management scheme for flash-based ssds. In *ACM TOS*, 2012.
[10] Park S., Jung D., and Kang J. et al. CFLRU: a replacement algorithm for flash memory. In *CASES*, 2006.
[11] Robinson J., and Devarakonda M. Data cache management using frequency-based replacement. In *SIGMETRICS*, 1990.
[12] Wu S., and Mao B. et al. Garbage collection aware cache management with improved performance for flash-based SSDs. In *ICS*, 2016.
[13] 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.
[14] 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.
[15] Du C., Yao Y., Zhou J., and Xu X. VBBMS: A novel buffer management strategy for NAND flash storage devices. In *IEEE TCE*, 2019.
[16] 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.
[17] Wang H., Yi X., and Huang P. et al. Efficient SSD caching by avoiding unnecessary writes using machine learning. In *ICPP*, 2018.
[18] Lacasa L., Luque B., and Ballesteros F. et al. From time series to complex networks: The visibility graph. In *PNAS*, 2008.
[19] 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.
[20] Narayanan D., Donnelly A., and Rowstron A. Write off-loading: Practical power management for enterprise storage. In *ACM TOS*, 2008.
[21] Lee C., and Matsuki T. et al. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *ACM SYSTOR*, 2017.