Frequent Access Pattern-based Prefetching Inside of Solid-State Drives

Xiaofei Xu*, Zhigang Cai*, Jianwei Liao*†, Yutaka Ishiakwa‡
*College of Computer and Information Science, Southwest University, Chongqing, China, 400715
†The State Key Laboratory for Novel Software Technology Nanjing University, Jiangsu 210023
‡RIKEN Center for Computational Science, Kobe, Japan, 650-0047
Corresponding author: Jianwei Liao, Email: *liaotoad@gmail.com*

***Abstract*—This paper proposes an SSD-inside data prefetching scheme, which has features of OS-dependence and use transparency. To be specific, it first mines frequent block access patterns that reflect the correlation among the occurred requests. Then it compares the requests in the current time window with the identified patterns, to direct fetching data in advance. Furthermore, to maximize the cache use efficiency, we construct a method to adaptively determine the cache partition on the basis of I/O workload characteristics, for separately buffering the prefetched data and the write data. Experimental results demonstrate that our proposal can yield improvements on average read latency by `6.3%` to `9.3%` without noticeably increasing write latency, in contrast to conventional SSD-inside prefetching schemes.**

***Index Terms*—SSD Cache, Frequent Access Pattern, Prefetching, Adaptive Cache Replacement, I/O Time.**

I. Introduction

The NAND flash memory-based solid-state drives (SSDs) are commonly employed in PCs, data centers and supercomputers, because of their features of small size, high performance, random-access performance and low energy consumption [3], [4], [14]. Apart from a NAND flash array that holds data, an SSD device generally has a micro-controller and a RAM memory. To be specific, the micro-controller runs Flash Translation Layer (FTL) to deal with logical-physical address mapping, garbage collection (GC) and wear-leveling (WL) [5], [11]. The RAM memory is used as the buffer inside of SSD[1] to cut down the number of write operations to the flash array, as well as keeping address mapping data structures [2], [22].

Data prefetching is a commonly used optimization scheme for disk based file systems, where fetching data from the disk dominates the overhead of read operations [17]. Specially, prefetching works well for target applications having regular access patterns on reads, such as database servers or scientific computations [17]. In an SSD setting, prefetching can mask read latency in flash data blocks, as the needed data were fetched to the RAM memory in advance. Consequently, it has been successfully applied to a variety of SSD-based storage systems [2], [9], [21].

Considering the memory size and processing power were limited in early SSDs, the main concern of existing prefetching schemes was to result in low cost and low power consumption [2]. As a result, most of them were involved with the operating system layer or even both of the operating system layer and SSDs [9], [21], which must damage the natures of compatibility and transparency. Although some inside SSD prefetching schemes have been proposed, their prediction models were generally confined to the limited resources, and can work for only a few application scenarios [2].

Nowadays SSDs have more computing power and memory capacity in micro-controllers than before. For example, the Cosmos OpenSSD platform [1], which is publicly released by the OpenSSD project, is equipped with more than `100MB` SDRAM and an embedded `1GHz` ARM CPU. We argue that designing a (near) universal prefetching scheme inside of SSDs becomes available, and such inside prefetching approaches do have advantages of OS-independence and use-transparency.

More importantly, the SSD cache is used to buffer not only the write data, but also the prefetched data if the prefetching functionality is enabled. On the one hand, conventional prefetching methods adopt a fixed size of cache or share the whole cache with write requests to buffer the prefetched data [2], [9], [21]. On the other hand, for boosting cache use efficiency, the division of write/read cache should depend on real-time read/write workloads and the cache hit accuracy.

To address the aforementioned issues, this paper proposes an SSD-inside prefetching mechanism, to better support data prefetching. In summary, it makes the following contributions:

- We propose a frequent access pattern-based prefetching scheme. It mines the frequent access patterns from the history of read requests, to guide prefetching data in the current time window. Moreover, we introduce an adaptive cache management policy for separately buffering the write data and the prefetched data in SSDs, on the basis of I/O workload characteristics.
- We conduct preliminary simulation evaluation by using `4` block traces of real world applications. As our measurements indicate, the newly proposed data prefetching mechanism can effectively reduce the average read latency without noticeably increasing write latency.

The rest paper is organized as follows: the background knowledge and related work are introduced in Section II. Section III specifically describes the proposed pattern-based prefetching scheme, and the adaptive cache partition management policy. Section IV presents the evaluation methodology

[1]The terms SSD RAM memory and SSD cache are used interchangeably in this paper.

978-3-9819263-4-7/DATE20/©2020 EDAA

and reports the experimental results. The paper is concluded in Section V.

II. Background and Related Work

This section briefly describes related work on data prefetching and other SSD-inside I/O optimization strategies.

Data prefetching and prediction models. For the purpose of achieving potential performance enhancements of storage systems, a variety of I/O history analysis-based I/O optimization mechanisms have been proposed [7], [12]. In fact, data prefetching has to predict future possible read requests to direct fetching data in advance. It means the accuracy of request prediction is critical to the effectiveness and applicability of data prefetching. Then, hidden Markov models, neural networks or other predictive algorithms are used to forecast I/O operations through analyzing I/O access patterns of the application [2], [6], [16], [21].

Prefetching approaches inside of SSD. Flashy prefetching aims to enhance prefetching effectiveness for SSDs [21]. Though this method runs at the application level, it still relies on operating systems to collect the I/O traces and manage the cached data. *Lynx* is another prefetcher for SSDs but running at OS layer in Linux kernel [9]. It employs Markov chains to forecast future read requests, to direct prefetching.

Xu et al. [23] argued that OS-dependent prefetching is lack of the features of use-transparency and compatibility. They have thus proposed an SSD-inside prefetching mechanism at FTL, without any modifications to OS or applications. It adopts a divide-and-conquer algorithm to reduce time and space complexity when conducting data prefetching, as they believe some SSDs may be resource-limited. However, nowadays SSDs are commonly equipped with powerful compute processing unit and considerable size of RAM.

SSD cache management for prefetched data. Considering both write data and prefetched data are buffered in SSD RAM, existing SSD prefetching schemes generally take advantage of a fixed cache division policy for buffering data. For instance, in the resource-optimized prefetcher, the size of the cache for holding the prefetched data is configured as `128KB` in evaluation experiments [23]. On the other side, different applications have varied read/write footprints, so that it does not make sense to allocate an unchanging part of cache for holding the prefetched data with respect to all cases.

SSD-inside Optimizations. A considerable number of studies exploit the computational power of SSD controller by offloading the data-intensive tasks to the embedded cores of SSDs [8], [19], [20]. For example, Jun et al. [8] take advantage of in-storage processing capacity to perform big data analytics, by exemplarily integrating the *Morris-Pratt (MP)* string search engine in SSD. In addition, Pei et al. [15] propose *Registor*, which aims to eliminate I/O bottlenecks in unstructured data processing that needs *regex* search. To this end, they have designed a hardware engine for *regex* search and deployed it inside of flash SSD, to deal with data on-the-fly during data transmission from SSD to host.

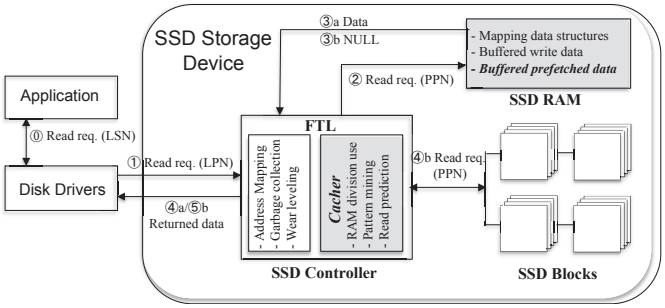

Fig. 1. High level overview of processing a read request in *Cacher-SSD*. It has two new features: ***(i)*** A software component of *Cacher* running at FTL, which is in charge of prefetching read data. ***(ii)*** the separate division of RAM dynamically adjusts to fit varied read/write workloads for holding the prefetched data.

III. Pattern-based SSD Prefetching

A. System Architecture

Figure 1 shows the high level overview of the proposed SSD-inside prefetching scheme, named as *Cacher-SSD*. In the case of cache is hit, the read request is satisfied with the data buffered in SSD RAM. Otherwise, the data stored in the flash data blocks will be read and forwarded to the application.

As seen, in addition to address mapping, garbage collection and wear-leveling, a new software component of *Cacher* runs at FTL. To be specific, *Cacher* deals with mining patterns, representing patterns and matching the identified patterns with the current requests, to direct data prefetching. Moreover, *Cacher* is in charge of cache space management, for separately buffering the write data and the prefetched data. We do not modify the garbage collection policy and the wear-leveling policy in this work and assume *Cacher-SSD* continues to employ the default ones.

B. Pattern-based Prefetching

1) Identifying and Modeling Frequent Access Patterns: The process of exploring access patterns can be addressed by the following two steps:

Step 1: The problem of mining the frequent patterns in our scenario can be described as follows: Let $T = \{a_1, a_2, ..., a_n\}$ be a set of logical addresses of n requests in the I/O track. The aim is to discover a collection of frequent patterns from the input I/O trace of T. A frequent access pattern can be expressed as $P_i = \{a_k, ..., a_x\}$, where all logical addresses of requests, such as a_k and a_x, appear multiple times in T.

We use a frequent item set mining approach called as the *FP-Growth* algorithm [18], to unearth frequent patterns by analyzing the requests in the previous time window. Consequently, we can acquire a number of frequent item sets, such as $\{Add_1, Add_2, Add_3, 4\}$. This exampled pattern has three frequently requested addresses, and the number of 4 is the minimum support, which implies all three addresses have been accessed with at least four times in all considered requests.

Step 2: After *Step 1*, it is possible to obtain many independent sets of access patterns. For the purpose of refining

Patterns \ LSNs	0	1	2	3	4	…	k-1	k	k+1	…	n
$\mathbf{P_0}${*0, 1, 3, k, k+1, n*}	1	0	0	0	0	…	0	1	1	…	0
$\mathbf{P_1}${*0, 2, 4, k-1, k+1*}	1	0	0	0	0	…	0	0	1	…	0
…	…										
$\mathbf{P_m}$ {*1, 3, 5, 7, k-1*}	0	1	0	0	0	…	0	0	0	…	0

Fig. 2. Matching matrix of requests and patterns. Note that we use `1` representing the corresponding logical sector number is in a specific pattern, and it has been accessed since the last processing round, and utilize `0` standing for other cases.

access patterns, we first sort them in descending order, by referring accessed time of addresses in patterns. Then, we check the access pattern that follows each occurrence of an identified pattern, and attempt to extend it. If more than half of elements in a specific pattern are also in another pattern, we carry out a union operation of two patterns, to form a new access pattern. Note that no extension to the access patterns should be performed if the number of access events in the extended pattern would exceed the upper limit.

2) Pattern Matching for Data Prefetching: The effectiveness of prefetching is primarily dependent on the prediction accuracy of future access requests [10]. The basic idea of pattern-based prefetching is to compare the current read requests with identified frequent patterns. In the case of a (major) part of addresses in a specific pattern have been accessed, it forecasts other remaining addresses in the same pattern are most likely to be requested in the near future. As a result, the relevant data of remaining addresses are supposed to be read in advance.

Furthermore, the speed of predictions on future requests is also critical to the effectiveness of the prefetching mechanism. For the purpose of accelerating the matching process, we have introduced a matrix to reflect the relationship between the logical sector numbers (*LSN*s) of requests and the identified patterns. As illustrated in Figure 2, there are `m` identified patterns, and each row of pattern shows its member elements (labeled as request *LSN*s). In the case of dealing with a read request, all elements in the corresponding column will be set as `1`, if its logical sector number is a part of the pattern.

Supposing the prefetching trigger condition is about more than a half of *LSN*s have been accessed in the pattern (i.e. the metric of *Matching hit threshold* in Table I of Section IV-A), and the data of remaining addresses are supposed to be fetched in advance. We take a case shown in Figure 2 as an example. The pattern of P_0 has `6` *LSN*s of *(0, 1, 3, k, k+1, n)*, in which *0, k* and *k+1* were requested. If the coming request targets at address of *n*, the process of data prefetching on *LSN*s of *1* and *3* will be activated, as a (major) part of addresses in this pattern have been accessed.

C. Adaptive Cache Management

For the purpose of improving the efficiency of cache use in SSD, we adaptively separate the cache for buffering the written data and the prefetched data respectively at different time windows of I/O processing.

The basic idea of adaptive cache management is to dynamically adjust the cache use on the basis of several impact factors. They are the numbers of read and write requests, the size of read and write space, and the numbers of read and write hits in cache in the previous time window.

First, we determine whether the prefetching functionality should be enabled or not after analyzing the statistical data of occurred I/O requests, by referring Equation 1. That is to say, the prefetching functionality is supposed to be dynamically disabled in specific time windows if τ is not less than a predefined value; otherwise, it will be supported.

$$\tau = \frac{W}{R} \times \frac{Prefetch_{miss}}{Prefetch_{all}} \tag{1}$$

where R and W are the numbers of total read and write requests in the previous time window. The parameters of $Prefetch_{miss}$ and $Prefetch_{all}$ indicate the numbers of non-hit prefetches and the total prefetches. Note that the minimum value of these parameters is fixed as `1`, though they might be `0` in some cases.

While prefetching is enabled, the proposed approach can make use of Equation 2 to calculate the ratio of SSD cache for holding the prefetched data.

$$\theta = \theta_{base} \times \frac{Size_R}{Size_W} + \frac{\alpha}{\beta} \times \frac{R-W}{R+W} \tag{2}$$

where θ_{base} is the base ratio of prefetch cache. $Size_R$ and $Size_W$ represent the read and write size in the previous time window. α and β denote the saved time caused by a read request being hit in the cache, and by a write request being hit in the cache. R and W are the numbers of occurred read and write requests in the previous time window.

In fact, α and β are two constants depending on platform configurations, $Size_R$, $Size_W$, R and W are four statistical values collected from the occurred requests. As a consequence, we may have varied sizes of cache (`0` for prefetching functionality disabled) for buffering the prefetched data at different time windows of running an application.

To achieve the goal of adaptive tuning the cache use for separately buffering the write data and the prefetched data, we evict either the cached write data or the prefetched data to load the new data, according to the partition ratio of cache use (i.e. θ_{cur}) in the current time window. Note that although we designate what type of data should be evicted, we still take advantage of the default policy, i.e. Least Recently Used (*LRU*) to practically ejected a corresponding page of data.

Algorithm 1 shows the specifications on the adaptive cache replacement scheme. In which, Line `7` identifies which part of cached data should be evicted. Specially, it compares the current cache ratio of prefetched data (i.e. θ_cur) to the one in the previous time window (i.e. θ_prev), to decide which kind of cached data should be replaced by the new data. Lines `8-13` present the details of dealing with a missed write request.

Algorithm 1: Adaptive cache replacement policy

Input: args of $cache\_size$, θ_cur, θ_prev;
Output: completion of I/O processing in the I/O queue;

```
1  /*0: replacing read pages, 1: replacing write pages*/
2  replace_target = 0;
3  /* processing requests in the current time window */
4  for req in I/O Queue do
5      if req hits in Cache then
6          continue;
7      replace_target = θ_cur ≥ θ_prev ? 1:0;
8      /*evicting cached data and buffering new write data*/
9      if req is a Write then
10         lru_replace(req → size, replace_target)
11         if !rep_flag then
12             θ_cur -= req → size/cache_size;
13         load_in_cache(req → data);
14     /*carrying out pattern-based data prefetching */
15     else if req hits in read_pattern then
16         pattern_prefetch(&buf, read_pattern);
17         lru_replace(buf → size, replace_target);
18         if rep_flag then
19             θ_cur += req → size/cache_size;
20         load_in_cache(buf → data);
21 /* preparing the ratio parameter for next round. */
22 θ_prev = θ_cur;
```

Lines 14-20 show a process of carrying out pattern-based prefetching.

IV. Experiments and Evaluation

A. Experiment Setup

We have performed trace-driven simulation with *SSDsim (ver2.1)*, which has a diverse set of configurations and supports of *TLC* flash simulation [24]. We have integrated our proposal with *SSDsim*, for supporting data prefetching inside of SSDs. Table I demonstrates our settings of *SSDsim* in experiments.

We employed 4 commonly used disk traces from the block I/O trace collection of Microsoft Research Cambridge [13]. The detailed specifications on the traces are shown in Table II. Besides, the following schemes are used in evaluation tests:

- *Baseline*: which indicates the feature of data prefetching is not supported. That is to say, the SSD cache is only used for buffering write data.
- *Lynx*: it employs *Markov* chains to forecast possible read requests in the future for directing data prefetching. The original *Lynx* runs at OS layer of Linux [9], but our *Lynx* implementation runs at FTL for comparison fairness.
- *Resource-Optimized Prefetching* (*ROP*): which also makes use of a *Markov* chains-based learning algorithm to predict batches of future reads, for eventually directing data prefetching [23]. But, a resource-optimized algorithm is used reduce the memory usage for holding the prefetched data. In other words, it will pick up what data of pages should be exactly prefetched after *Markov* chains prediction.
  We argue that *ROP* might be the most related work to ours, as it is implemented inside of SSDs and has the features of OS-dependence [23].
- *Pattern*: which is the newly proposed prefetching scheme. It enables frequent pattern-based data prefetching inside of SSDs and supports adaptive cache partition for separately bufferring the write data and the prefetched data.

TABLE I
Experimental settings of *SSDsim* (TLC cell)

Parameters	Values	Parameters	Values
Capacity	32GB	*Read time*	0.075ms
Page per block	192	*Write time*	2.67ms
Page size	8K	*Cache read/write*	0.001us
Cache size	32MB	*Elements in pattern*	[4, 10]
Fixed prefetched cache	8%	*Matching hit threshold*	50%

Note: Parameters of α and β in Equation 2 are then configured as 0.074ms and 2.669ms.

TABLE II
Specifications on Selected Disk Traces

Trace	# of Req.	Read ratio	Read size	Read/All footprint
hm_0	3993316	35.5%	7.4KB	2.72/7.64GB
usr_0	2237889	40.4%	40.9KB	2.32/3.96GB
$src2\_2$	1156885	30.3%	68.1KB	20.67/54.49GB
mds_0	1211034	11.9%	40.9KB	2.93/3.82GB

The number of I/O requests in each time window is setting to 1024. And we select the first 256 requests in each window to generate frequent patterns to guide prefetching. After certain preliminary tests and sensitive analysis, we set the predefined threshold for τ, which is used for enabling or disabling the prefetching functionality, as 0.1. In addition, the base ratio of prefetching cache shown in Equation 2 is configured as 0.15 in our tests.

B. Results and Discussion

1) Read Response Time: Figure 3 shows the average read latency of the selected block traces, while using different prefetching schemes and *Baseline*. As seen, three prefetching schemes can achieve an improvement on average read response time in a major part of traces. Specifically, the proposed *Pattern* can reduce the read latency by 9.2%, 6.3%, and 9.3% on average, in contrast to *Baseline*, *Lynx*, and *ROP*.

Another noticeable clue shown in Figure 3 is about all prefetching schemes do not outperform *Baseline*, in the case of processing *hm_0*. This is because in the *hm_0* trace, 60.1% hot read addresses are also the hot write address[2], so that prefetching cannot offset the overhead of moving some write data out of the cache. In addition, *hm_0* consists of many small

[2]In this case, we define hot access addresses if they have been requested not less than 4 times.

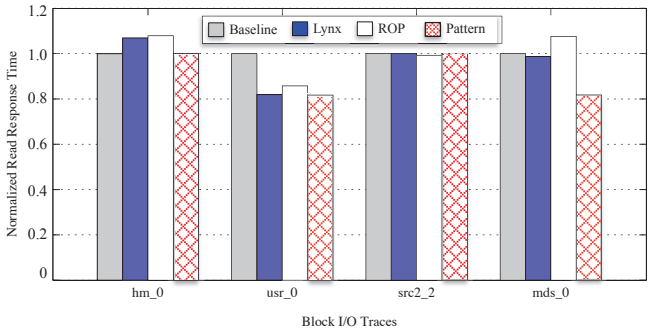

Fig. 3. Average read latency of selected block traces.

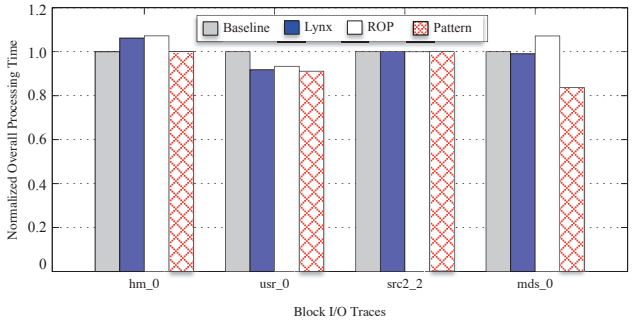

Fig. 4. The total processing time of running selected block traces.

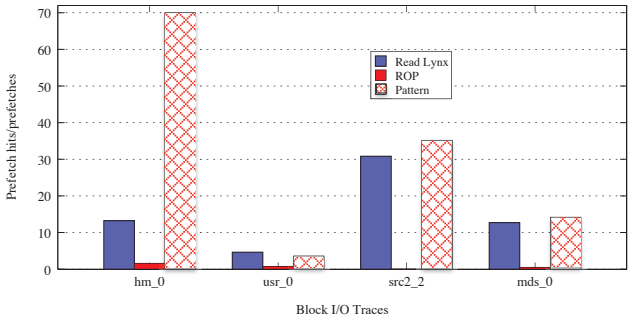

Fig. 5. The results of prefetching hits per prefetch.

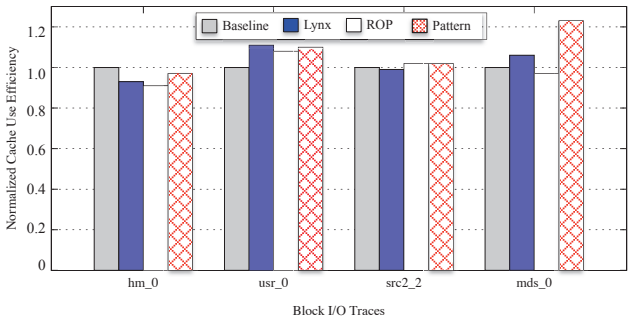

Fig. 6. Cache use efficiency by representing with total cache hits.

read requests, whose size is even less than the prefetching unit, i.e. a page of 8K, that might be detrimental to prefetching.

But, it is worth to mention that the *Pattern* approach yields a very similar read latency to *Baseline*, as it is able to adaptively cut down the capacity of prefetch cache to 0 and all cache is used for buffering write data.

2) Overall I/O Time: Buffering the prefetched data in SSD cache must place negative effects to the performance of write data, as a part of cache is separated for the prefetched data. We then record the overall I/O time to show the effectiveness of data prefetching, and Figure 4 presents the results.

Clearly, they are very similar to the results of read latency after running the block traces. In other words, the proposed *Pattern* approach outperforms others when processing the traces excluding *hm_0*.

3) Prefetching hit per prefetch: To measure the effectiveness of different prefetching schemes, we record results of *prefetching hits* per *prefetch*, which is the division of the total prefetching hits by the prefetch count. The total prefetching hit indicates the number of read on the prefetched data in the unit of page, and the prefetch count means the number of prefetched pages.

Figure 5 shows the results of prefetching hits per prefetch by using three prefetching approaches. As demonstrated, the proposed *Pattern* scheme yields the best outcomes, compared with other two related work. On the other side, *ROP* performs the worst, this is because it prefetches the largest number of pages in our tests, and the average hit on the prefetched data becomes unattractive.

4) Cache Use Efficiency: We define the cache use efficiency as the sum of the total read cache hits multiplied by the benefit of read hit (i.e. the read time - the read hit time), and the total write cache hits multiplied by the benefit of write hit (i.e. the write time - the write hit time). As seen in Figure 6, the proposed scheme can improve cache use efficiency by up to 21.1%, compared with other two prefetching schemes.

5) Analysis on Adaptive Cache Partition: *Lynx* and *ROP* employ a fixed 8% cache for buffering the prefetched data. On the other side, adaptive cache partition aims to allotting an appropriate part of cache for holding the prefetched data, by referring the characteristics of I/O workloads and the prefetching productivity. According to our design, we will turn off the prefetching functionality if it may not bring about positive effects, that indicates the ratio is 0.

Table III presents the statistics on the ratio of prefetch cache to all cache. As shown, it takes advantage of different ratios for different traces, and even varied ratios in different time windows for the same trace.

C. Summary

With respect to comparing pattern-based prefetching and Markov chains-based prefetching, we emphasize the following two key observations. ***First***, the proposed pattern-based prefetching scheme can cause a higher degree of accuracy in forecasting a batch of future read requests, by avoiding unnecessary prefetching operations. ***Second***, the scheme of adaptive cache management can dynamically adjust the cache

TABLE III
THE STATISTICS ON THE RATIO OF PREFETCH CACHE TO ALL CACHE

	Max ratio	Min ratio (Non-zero)	Avg. ratio	STDEV
hm_0	3.9%	0.0% (2.1%)	0.5%	0.011
usr_0	9.0%	7.0% (7.0%)	7.9%	0.005
$src2\_2$	11.0%	0.0% (9.0%)	6.0%	0.051
mds_0	10.7%	6.1% (6.1%)	8.6%	0.025

partition for the prefetched data and the write data, to boost the efficiency of cache use. In brief, we conclude that the proposed prefetching scheme is able to significantly reduce the time required by conducting data prefetching.

V. CONCLUSIONS

This paper has proposed, implemented, and evaluated an OS-independent data prefetching mechanism for SSDs. It first mines frequent read patterns from the history of read accesses. After that, a matrix data structure is introduced to match the in-queue read requests and the mined patterns, for guiding data prefetching. Furthermore, in order to maximize the use efficiency of SSD cache, we have proposed an adaptive cache partition scheme, to separately buffer the write data and the prefetched data, on the basis of the write/read scale of workload and prefetching productivity.

Through a series of emulation experiments based on several realistic disk traces, we show that the proposed pattern-based prefetching scheme can reduce I/O response time by up to 9.1% on average. Besides, the experimental results illustrate the adaptive cache partition policy has the nature of flexibility, and can noticeably enhance the cache use efficiency by up to 21.1%, in contrast to other comparison counterparts.

ACKNOWLEDGMENT

This work was partially supported by "National Natural Science Foundation of China (No. 61872299, 61732019, 61672435, 61811530327)", "the Capacity Development Grant of Southwest University (No.SWU116007)", "Natural Science Foundation Project of CQ CSTC (No. CSTC2018jcyjAX0552, CSTC2017jcyjAX0295)", and "the Opening Project of State Key Laboratory for Novel Software Technology (No. KFKT2019B06)".

REFERENCES

[1] Cosmos OpenSSD Platform. http://www.openssd-project.org.
[2] Cui J., Zhang Y., and Huang J. et al. ShadowGC: Cooperative garbage collection with multi-level buffer for performance improvement in NAND flash-based SSDs. In proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE '2018), pp. 1247-1252. 2018.
[3] Gao C., Shi L., and Di Y. et al. Exploiting Chip Idleness for Minimizing Garbage CollectionInduced Chip Access Conflict on SSDs. ACM Transactions on Design Automation of Electronic Systems, Vol. 23(2): 15, 2018.
[4] Grupp L., Davis J., and Swanson S. The bleak future of NAND flash memory. In proceedings of the 10th USENIX conference on File and Storage Technologies (FAST '2012), 2012.
[5] Gupta A., Pisolkar R., and Urgaonkar B. et al. Leveraging Value Locality in Optimizing NAND Flash-based SSDs. In proceedings of the 9th USENIX conference on File and Storage Technologies (FAST '2011), 2011.
[6] He J., Bent J., Torres A., and Sun X. I/O Acceleration with Pattern Detection. In proceedings of the 22nd international symposium on High-performance parallel and distributed computing (PPoPP '2013), pp. 25-36, 2013.
[7] Jiang S., Ding X., Xu Y., and Davis K. A Prefetching Scheme Exploiting both Data Layout and Access History on Disk. ACM Transactions on Storage, Vol. 9(3): 10, 2013.
[8] Jun S., Liu M., and Lee S. et al. Bluedbm: An appliance for big data analytics. In proceedings of ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA '2015), pp. 2-13, 2015.
[9] Laga A., Boukhobza J., and Koskas M. et al. Lynx: A learning linux prefetching mechanism for ssd performance model. In proceedings of 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA '2016), pp. 1-6, 2016.
[10] Li Z., Chen Z., Srinivasan S., and Zhou Y. C-Miner: Mining Block Correlations in Storage Systems. Proceedings of the 3rd Conference on File and Storage Technologies (FAST '2004), pp. 173-186, 2004.
[11] Liao J., Zhang F., and Li L. et al. Adaptive wear-leveling in flash-based memory. IEEE Computer Architecture Letters, Vol. 14(1): 1-4, 2015.
[12] Liao J., Trahay F., and Gerofi B. et al. Prefetching on storage servers through mining access patterns on blocks. IEEE Transactions on Parallel and Distributed Systems, Vol. 27(9): 2698-2710, 2016.
[13] Narayanan D., Donnelly A., and Rowstron A. Write off-loading: Practical power management for enterprise storage. ACM Transactions on Storage, Vol. 4(3): 10, 2008.
[14] Matsui C., Sun C., and Takeuchi K. Design of hybrid SSDs with storage class memory and NAND flash memory. Proceedings of the IEEE, Vol. 105(9): 1812-1821, 2017.
[15] Pei S., Yang J., and Yang Q. REGISTOR: A platform for unstructured data processing inside SSD storage. ACM Transactions on Storage, Vol. 15(1): 7, 2019.
[16] Peled L., Weiser U., and Etsion Y. Towards Memory Prefetching with Neural Networks: Challenges and Insights. *arXiv preprint* arXiv:1804.00478, 2018.
[17] Shriver E., Small C., and Smith K. Why does file system prefetching work? In proceedings of USENIX Annual Technical Conference (ATC '1999), pp. 71-84, 1999.
[18] Tan P., Steinbach M., and Karpatne A. et al. Introduction to data mining. Pearson Education India, 2007.
[19] Tiwari D., Boboila S., and Vazhkudai S. et al. Active flash: towards energy-efficient, in-situ data analytics on extreme-scale machines. In proceedings of the 11th USENIX conference on File and Storage Technologies (FAST '2013), 2013.
[20] Tseng H., Zhao Q., and Zhou Y. et al. Morpheus: creating application objects efficiently for heterogeneous computing. In proceedings of ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA '2016), pp. 53-65, 2016.
[21] Uppal A., Chiang R., and Huang H. Flashy prefetching for high-performance flash drives. In proceedings of IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST '2012), pp. 1-12, 2012.
[22] Wang Y., Kim K., and Lee B. et al. A novel buffer management scheme based on particle swarm optimization for SSD. The Journal of Supercomputing, Vol. 74(1): 141-159, 2018.
[23] Xu R., Jin X., and Tao L. et al. An efficient resource-optimized learning prefetcher for solid state drives. In proceedings of Design, Automation & Test in Europe Conference & Exhibition (DATE '2018), pp. 273-276. 2018.
[24] 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 proceedings of the 2018 International Conference on Supercomputing (ICS '2018), pp. 22-32, 2018.