Pattern-Based Prefetching with Adaptive Cache Management Inside of Solid-State Drives

JUN LI, XIAOFEI XU, ZHIGANG CAI, and JIANWEI LIAO, Southwest University of China, China
KENLI LI, Hunan University, China
BALAZS GEROFI and YUTAKA ISHIKAWA, RIKEN Center for Computational Science, Japan

This article proposes a pattern-based prefetching scheme with the support of adaptive cache management, at the flash translation layer of **solid-state drives** (**SSDs**). It works inside of SSDs and has features of OS dependence and uses transparency. Specifically, it first mines frequent block access patterns that reflect the correlation among the occurred I/O requests. Then, it compares the requests in the current time window with the identified patterns to direct prefetching data into the cache of SSDs. More importantly, to maximize the cache use efficiency, we build a mathematical model to adaptively determine the cache partition on the basis of I/O workload characteristics, for separately buffering the prefetched data and the written data. Experimental results show that our proposal can yield improvements on average read latency by 1.8%–36.5% without noticeably increasing the write latency, in contrast to conventional SSD-inside prefetching schemes.

CCS Concepts: • **Computer systems organization** → **Embedded software;**

Additional Key Words and Phrases: SSDs, SSD cache, prefetching, frequent access pattern, adaptive cache management, I/O time

ACM Reference format:
Jun Li, Xiaofei Xu, Zhigang Cai, Jianwei Liao, Kenli Li, Balazs Gerofi, and Yutaka Ishikawa. 2022. Pattern-Based Prefetching with Adaptive Cache Management Inside of Solid-State Drives. *ACM Trans. Storage* 18, 1, Article 7 (January 2022), 25 pages.
https://doi.org/10.1145/3474393

This is a revised version. A preliminary version of this work was published in the Proceedings of 2020 Design, Automation & Test in Europe Conference & Exhibition (DATE'20).
J. Liao works for Southwest University of China, and State Key Lab. for Novel Software Technology, Nanjing University, People's Republic of China.
This work was partially supported by National Natural Science Foundation of China (No. 61872299, No. 62032019), Chongqing Graduate Research and Innovation Project (No. CYB21110), Chongqing Talent (Youth, No. CQYC202005094), and the Opening Project of State Key Laboratory for and Novel Software Technology (No. KFKT2021B06).
Authors' addresses: J. Li, X. Xu, Z. Cai, and J. Liao (corresponding author), Southwest University of China, Chongqing 400715, China; emails: {junli95, nakamura}@email.swu.edu.cn, czg@swu.edu.cn, liaojianwei@il.is.s.u-tokyo.ac.jp; K. Li, Hunan University, Hunan 410082, China; email: lkl@hnu.edu.cn; B. Gerofi and Y. Ishikawa, RIKEN Center for Computational Science, Kobe 650-0047, Japan; emails: bgerofi@riken.jp, ishikawa@is.s.u-tokyo.ac.jp.

1553-3077/2022/1-ART7 $15.00
https://doi.org/10.1145/3474393

1 INTRODUCTION

The NAND flash memory–based **solid-state drives** (**SSDs**) are commonly employed in PCs, data centers, and supercomputers, because of their virtues of small size, high data throughput, random-access performance, and low energy consumption [1–3]. Apart from a NAND flash array that holds data, an SSD device generally has a micro-controller and a **Random Access Memory** (**RAM**). 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**) [4–7]. 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 [8, 9].

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 [12]. Specifically, prefetching works well for target applications having regular access patterns on reads, such as database servers or some scientific computations [12, 17]. In an SSD setting, prefetching can mask read latency in flash data blocks, as the needed data were loaded into the RAM memory in advance. Consequently, it has started to be applied to a variety of SSD-based storage systems [13, 14, 20]. Note that the prefetching methods in the file system may run at the levels of virtual file system and file system, but the prefetching schemes inside of SSDs run at the level of SSD devices in the I/O stack.

Considering the memory size and processing power were limited in early SSDs, the main concerns of existing prefetching schemes were about the low computation cost and the low power consumption [20]. Then, most of them were involved with the operating system layer or even both the operating system layer and SSDs [13, 14], 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 computation and memory resources [20].

We say that nowadays SSDs have large computing power in micro-controllers as well as more memory capacity inside. For example, the Cosmos OpenSSD platform [15], which is publicly released by the OpenSSD project, is equipped with more than `100 MB` SDRAM and an embedded `1 GHz` ARM CPU. Then, we argue that designing a (near) universal prefetching scheme inside of modern SSDs becomes available, and such inside prefetching approaches do have advantages of OS independence and use transparency. That is to say, such SSDs can be deployed in any context, as their micro-controllers are in charge of prefetching tasks.

In the case in which the prefetching functionality is enabled, the SSD cache is used to buffer not only the written data, but also the prefetched data. Conventional prefetching methods adopt a fixed portion of cache to buffer the prefetched data [13, 14, 20]. However, such fixed cache separation schemes cannot fit all I/O characteristics in varied applications. For example, the prefetching cache is not expected when running write-dominant workloads, but the cache for the prefetched data are supposed to be enlarged while executing read-intensive applications in general. More specifically, some applications may change their read/write workloads during the lifetime, which indicates that data prefetching may work in some time windows of lifetime, but fail in other time windows. Then, in order to boost cache use efficiency, the division of write/prefetch cache[2] should depend on real-time factors in the workloads such as the read/write ratio and the prefetching accuracy.

To address the aforementioned issues, this article proposes an SSD-inside prefetching mechanism with adaptive cache management, to better improve I/O performance of SSDs. In summary, it makes the following contributions:

[1] The terms SSD RAM memory and SSD cache are used interchangeably in this article.
[2] The terms write cache and prefetch cache represent the cache space for buffering the written data and the prefetched data, respectively.

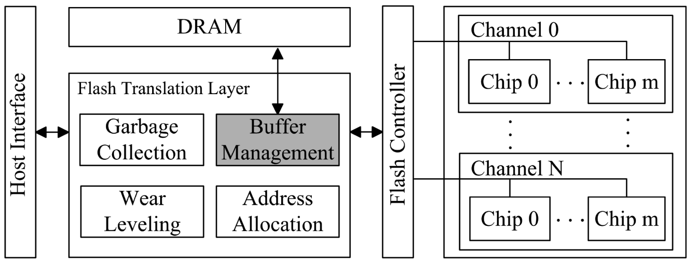

Fig. 1. The internal architectural overview in SSDs.

(1) We propose a frequent access pattern–based prefetching scheme. It first mines the frequent access patterns from the history of read requests to stand for the address collections that are frequently read together. Then, it holds a matrix to maintain the pattern information and to carry out the pattern matching for directing data prefetching in the current time window.
(2) We construct a mathematical model by considering several comprehensive factors, such as the read/write ratio and the historical prefetching accuracy, to support adaptive cache management. Then, it is able to more precisely adjust the partition of the prefetch cache and the write cache from case to case.
(3) We offer preliminary evaluation on several disk traces of real-world applications. Apart from illustrating the performance benefits and scalability of our proposal, we specifically measure its time and space overhead and then assess their performance impacts.

Note that, as an extension of our previous work [35], only the last two contributions are new in this article. As our measurements indicate, the newly proposed data prefetching mechanism with adaptive cache management can further reduce the average read latency without noticeably increasing the write latency.

The rest of the article is organized as follows: the background knowledge and related work are introduced in Section 2. Section 3 specifically describes the proposed pattern-based prefetching scheme, and the adaptive cache partition management policy. Section 4 presents the evaluation methodology and reports the experimental results. Finally, the article is concluded in Section 5.

2 BACKGROUND AND MOTIVATION

2.1 SSD Architecture

Figure 1 shows an internal architectural overview of SSDs, including software and hardware components. Obviously, the main software layer of SSD is FTL, which takes charge of address mapping, garbage collection [9], wear-leveling [11], and buffer management. Specifically, the buffer management module running at FTL controls the use of **dynamic random access memory** (**DRAM**) of SSDs. In other words, not only the data structures of the address mapping table are saved in the buffer, but also the contents of write requests are temporarily cached in there to minimize the latency of responding write requests [8, 34].

2.2 Related Work

This section will briefly summarize the related work on data prefetching, SSD cache management, and other SSD-inside I/O optimization strategies.

Data prefetching and prediction models. In order to yield potential performance enhancements of storage systems, a variety of I/O history analysis-based I/O optimization mechanisms have been proposed and evaluated [16, 17]. In fact, data prefetching has to predict future possible read requests to direct fetching data in advance. That is, 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 [13, 18–20].

Prefetching approaches focusing on SSD. Flashy prefetching aims to enhance prefetching effectiveness for SSDs [13]. Though this method runs at the application level, it still relies on the **operating system** (**OS**) to collect the I/O traces and manage the cached data. *Lynx* is another prefetcher for SSDs but running at the OS layer in the Linux kernel [14]. It makes use of Markov chains to forecast future read requests, to guide reading the relevant data in advance.

Xu et al. [20] argued that OS-dependent prefetching lacks the advantages 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 purposely reduce time and space complexity when conducting data prefetching, as they believe some SSDs may be resource-limited.

Considering modern SSDs are commonly equipped with a powerful compute processing unit and considerable size of RAM, we have proposed a prefetching approach to more accurately prefetch the expected data in our previous work [35]. In this method, we take advantage of the powerful SSD micro-controller to mine the frequent access patterns, which reflect the correlation among the occurred requests. Then, the process of prefetching data will be only triggered if a mined pattern is hit in the current time window.

SSD cache management for prefetched data. Cache approaches are mainly designed for RAM embedded in SSD (i.e., buffer cache). They generally intend to achieve cache hit rate improvement for maximizing the efficiency of cache usage. Considering both written data and prefetched data are buffered in SSD RAM, existing SSD prefetching schemes generally take advantage of a fixed cache division policy for buffering two kinds of data. For instance, in the resource-optimized prefetcher, the size of the prefetch cache is configured as 128 KB in evaluation experiments [20].

But, different applications have varied read/write footprints, so that it does not make sense to allocate an unchanging part of prefetch cache with respect to all cases. To address this issue, Xu et al. [35] have presented an empirical formula to divide the SSD cache into prefetch cache and write cache by separately holding the prefetched data and the written data, by referring to the read/write ratio in the applications.

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 [21–23]. For example, Jun et al. [23] 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. [24] 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.

2.3 Motivations

We have analyzed certain block I/O traces of real-world applications, to disclose their probability distribution of read frequency on the address space and their number of hot read addresses at different execution stages.

Figure 2 illustrates the results of probability distribution of three sampled applications in the Microsoft Research Cambridge block I/O trace collection [31]. They are *hm_1* (*read intensive*), *usr_0* (*read-write balanced*), and *src1_2* (*write intensive*). In the figure, the X-axis scales from 0 to 100%, which represents the proportion of read addresses ordering by their read counts, and the Y-axis

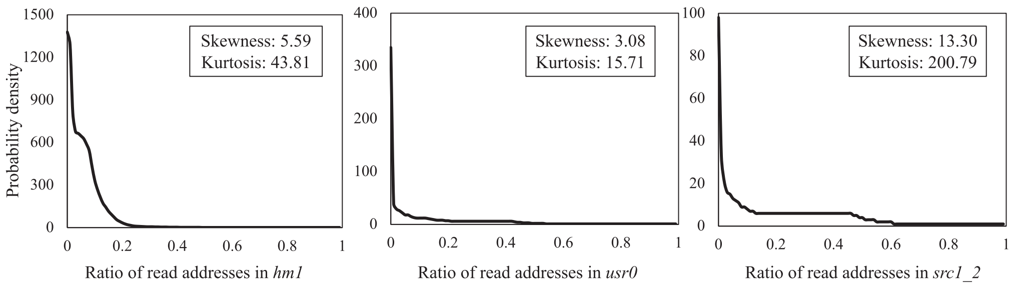

Fig. 2. The probability density of read frequency on the address space of some traces in the MSRC block trace collection [31] (spatial locality of block accesses).

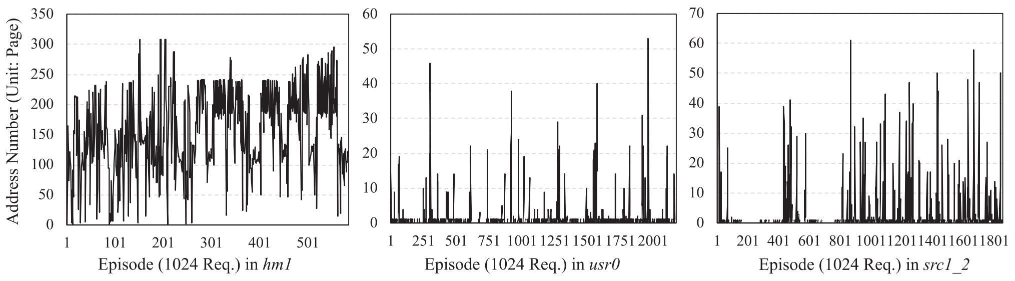

Fig. 3. The frequent read address space (i.e., not less than three) number in the whole application period (temporal locality of block accesses).

denotes the probability density of the read requests. Obviously, all block I/O traces show the locality of data access. That is to say, only a very small part of data (e.g., the left part of the X-axis in the figure) is intensively accessed after running the applications. Moreover, in order to characterize the location and variability of read frequency on block addresses in the traces, we calculate the measures of skewness and kurtosis [25] for better illustrating the access locality. As seen in the figure, these numerical values show that the read counts of block addresses are lack of symmetry and heavy-tailed. Then, we argue that only a small number of addresses have intensive read workloads.

Observation I. *Applications read their data and obey the reference of space locality. Thus, prefetching the hot read data into the SSD buffer may contribute to better I/O performance caused by read hits.*

Figure 3 shows the number of frequent read addresses in the lifetime of applications. The X-axis represents execution stages (each of them is composed of 1,024 requests), and the Y-axis shows the number of frequent accessed (i.e., not less than three) addresses. The results reveal that the hot read addresses differ from varieties of applications, and keep changing during different stages after replaying the selected block I/O traces. More importantly, this figure also reveals that certain read addresses will be intensively read in a short time period, which verifies the block accesses have temporal locality.

Observation II. *The number of hot read addresses keeps changing at varied running stages in applications. Thus, dynamically adjusting the size of prefetch cache may boost the cache use efficiency.*

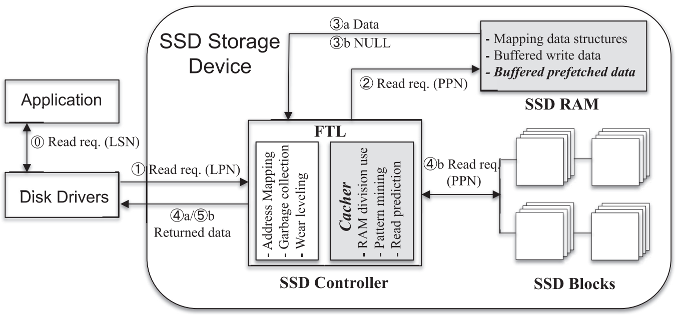

Fig. 4. High level overview of processing a read request in *Cacher-SSD*. It has two new features: (i) A software component of *Cacher* running at FTL, which is in charge of prefetching read data. (ii) The separate division of RAM dynamically adjusts to fit varied read/write workloads for holding the prefetched data.

Such observations motivate us to efficiently prefetch data into the SSD buffer by the following: (1) prefetching the frequently read data may increase prefetching hits and then boost read performance; (2) adjusting the size of the prefetch cache in an adaptive manner can contribute to better SSD buffer utilization.

3 PATTERN-BASED SSD PREFETCHING WITH ADAPTIVE CACHE MANAGEMENT

3.1 System Architecture

Figure 4 shows the high level overview of the proposed SSD-inside prefetching scheme, and we name such SSD as *Cacher-SSD*. In the case of a cache hit, the read request can be satisfied with the prefetched data buffered in SSD RAM. Otherwise, the data stored in the flash data blocks will be read and then 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. Specifically, *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 separating the write cache and the prefetch cache. 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.

Figure 4 also illustrates the process of data prefetching in SSD devices. When a read request comes from an application, SSD controller receives the request and first searches the relevant data in the SSD RAM. If it hits in RAM buffer, then the cached data directly responds to the application. Otherwise, SSD controller tries to read the data from SSD blocks, and replies the data to the application. That is, if the prefetched data can be directly sent to the application, the read response time can be significantly cut down, but we must bear the prefetching overhead if the prefetched data are never hit.

3.2 Pattern-Based Prefetching

To precisely direct data prefetching, we propose pattern-based prefetching. Specifically, we refer to the correlation of data addresses as a request pattern, which is a special set of address items (that may be accessed together in a short period). Then, while some addresses in the pattern have been requested, the proposed scheme will fetch the data in the remainder addresses into the SSD cache in advance, as it believes other addresses in the pattern are most likely to be accessed soon.

LSNs / Patterns	**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. 5. 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.

3.2.1 *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 $T = \{a_1, a_2, \ldots, a_n\}$ 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 $P_i = \{a_k, \ldots, a_x\}$, where all logical addresses of requests, such as a_k and a_x, appear multiple times in T.

We take advantage of a widely used frequent item set mining approach called as the *FP-Growth* algorithm [26], to unearth frequent patterns by analyzing the requests in the previous time window. To be specific, the *FP-Growth* algorithm takes advantage of a divide-and-conquer strategy to extract frequent item sets without using candidate generations, so that it is efficient and scalable for mining both long and short frequent patterns. The core of this approach is the usage of a special data structure named **frequent-pattern tree** (***FP-tree***), which retains the itemset association information [26]. Consequently, by utilizing *FP-Growth*, we can acquire a number of frequent item sets, such as $\{Add_1, Add_2, Add_3, 4\}$. This example pattern has three frequently requested addresses, and the number of 4 is the minimum support, which implies all three addresses have been accessed 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 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.

3.2.2 *Pattern Matching for Data Prefetching.*

The effectiveness of prefetching is primarily dependent on the prediction accuracy of future access requests [27]. The basic idea of pattern-based prefetching is to compare the current read requests with identified frequent patterns. In the case in which 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 5, there are `m` identified patterns, and each row of patterns shows its member elements (labeled as request *LSN*s). Note that the set of columns is the

LSN union of the mined patterns in a given time window. 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 1 of Section 4.1), and the data of remaining addresses are supposed to be fetched in advance. We take a case shown in Figure 5 as an example. The pattern of P_0 has six *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.

3.3 Adaptive Cache Management

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 to 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.

$$\tau = \frac{W_{num}}{R_{num}} \times \frac{Prefetch_{miss}}{Prefetch_{all}}, \tag{1}$$

where R_{num} and W_{num} 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 both of $Prefetch_{miss}$ and $Prefetch_{all}$ will be assigned as 1 if they are less than 1.

For the purpose of improving the efficiency of cache use in SSD, this section presents an adaptive cache partition policy that dynamically divides the write cache and the prefetch cache at different time windows of I/O processing. In our previous work [35], we proposed an adaptive cache division policy by referring to I/O characteristics of workloads, but it does not take the I/O frequency and address distribution into consideration.

In this section, we further build a mathematical model to estimate the return on cache use introduced by data prefetching, called *Pattern+*. Equation (2) demonstrates the overall return of prefetching data, consisting of the saved read response time (i.e., T_1) and the prolonged write latency due to write cache misses (i.e., T_2).

$$T = T_1 - T_2. \tag{2}$$

On the one side, T_1 is positively correlated with the total number of read requests that are hit in the cache. Then, it can be illustrated as Equation (3).

$$T_1 = \alpha \times \gamma \times h, \tag{3}$$

where α, γ, and h denote the saved time caused by a read request being hit in the cache, the number of occurred read requests, and the proportion of read hits in the cache, respectively.

Specifically, the parameter of h depends on factors of the total size of prefetched data and what data have been prefetched. Section 2.3 discusses the probability distribution of read frequency on the address space of applications, which verifies only a small part of data are intensively accessed. Then, we build the model for adaptively adjusting the size of the prefetch cache, by referring to the probability distribution of block accesses. As seen in Figure 6(a), the X-axis scales from `0` to `100%`, which represents the proportion of read addresses ordering by their read counts, and the

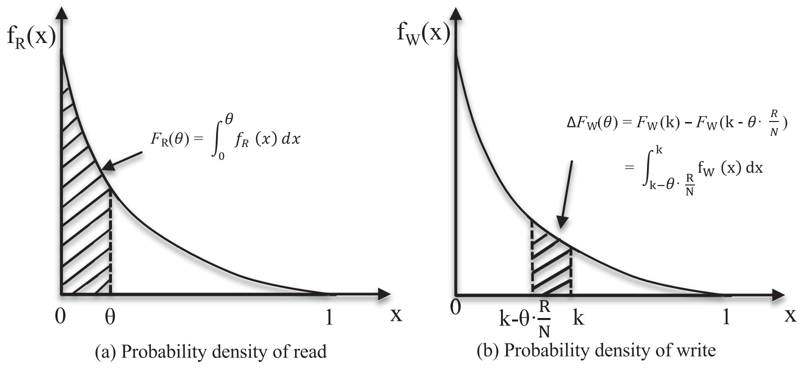

Fig. 6. The probability density of (a) read address frequency and (b) write address frequency.

Y-axis denotes the probability density of the read requests. That is to say, $F_R(x) = \int_0^1 f_R(x)dx = 1$. Figure 6(b) demonstrates the case of write requests, which is same as the read case.

Assume the θ part of total read pages of data have been prefetched, shown in Figure 6(a), which are expected to be frequently accessed in the near future. The expression of $F_R(\theta) = \int_0^\theta f_R(x)dx$ represents the ratio of reads on the θ part of read data to the total number of reads. Considering the θ part of read data are prefetched, the saved read response time (i.e., T_1) can be quantified by Equation (4).

$$T_1 = \alpha \times \gamma \times F_R(\theta). \tag{4}$$

On the other hand, the prefetched data must occupy the cache space, which was supposed to be used for buffering the written data, and is also shown in Figure 6. The initial ratio of the cached written data to all data is k, and θ of read data are supposed to be prefetched, so that the reduction of probability of write hit can be expressed:

$$\begin{aligned}\Delta F_W(\theta) = F_W(k) - F_W\left(k - \theta \cdot \frac{R}{N}\right) &= \int_{k-\theta\cdot\frac{R}{N}}^{k} f_W(x)dx \\ &= \theta \cdot \frac{R}{N} \times \bar{f}_W\left(\theta \cdot \frac{R}{N}\right) \\ &\approx \theta \cdot \frac{R}{N} \times f_W(k),\end{aligned} \tag{5}$$

where $F_W(k)$ means the ratio of the writes on the most frequently accessed k cached pages of written data to the total number of write operations. R and N represent the read address space and the total address space, respectively. The parameter of $\bar{f}_W(\theta \cdot \frac{R}{N})$ is the average of write frequency to the pages within the range of $(k - \theta \cdot \frac{R}{N}, k)$, and $f_W(k)$ is the write frequency to the page having the minimum access count in the range of $(k - \theta \cdot \frac{R}{N}, k)$.

In fact, $f_W(k)$ indicates the minimum hit ratio of the cached written data; it will decrease when the hit ratio of cached written data degrades. As a consequence, the negative effect of prefetching can be defined as Equation (6).

$$T_2 = \beta \times \omega \times \theta \cdot \frac{R}{N} \times \bar{f}_W\left(\theta \cdot \frac{R}{N}\right), \tag{6}$$

where β is the saved time by buffering a page of written data, and ω is the number of occurred write requests.

By referring back to Equation (5), the overall benefit of prefetching the most frequently accessed θ part of read data can be measured by using Equation (7).

$$
\begin{aligned}
T_\theta &= \alpha \times \gamma \times F_R(\theta) - \beta \times \omega \times \theta \cdot \frac{R}{N} \times \bar{f}_W\left(\theta \cdot \frac{R}{N}\right) \\
&\approx \alpha \times \gamma \times F_R(\theta) - \beta \times \omega \times \theta \cdot \frac{R}{N} \times f_W(k).
\end{aligned}
\tag{7}
$$

According to the first-order condition of $\frac{dT}{d\theta} = 0$, and $f'(\theta) < 0$, it can be concluded that T_θ will reach the maximum value in the case in which the condition of $f(\theta^*) = \frac{\beta}{\alpha} \cdot \frac{\omega}{\gamma} \cdot \frac{R}{N} \cdot f_W(k)$ holds. In fact, α and β are two constants depending on platform configurations; R, N, γ, and ω are four statistical values collected from the occurred requests. The value of $f_W(k)$ is the smallest write count to the buffered written data in the last time window.

The value of $f(\theta^*)$ can be leveraged to split SSD cache for keeping the prefetched data. To this end, the read pages are sorted in descending order by referring their access counts. Then, the read data whose read counts are not less than $R \cdot f(\theta^*)$, should be initially loaded into the cache.

Then, we may have varied sizes of cache for buffering the prefetched data at different time windows. To achieve the goal of adaptive tuning the cache use for separately buffering the written data and the prefetched data, we evict either the cached written 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.

Algorithm 1 shows the specifications on the adaptive cache replacement scheme. As read, line 11 identifies which part of cached data should be evicted. Specifically, it compares the current read pages (i.e., *rd_pages_cur*) to the read pages in the previous time window (i.e., *rd_pages_prev*), to determine which kind of cached data should be replaced by the new data. Lines 13–19 present the details of dealing with a missed write request. Lines 21–28 show the process of carrying out pattern-based prefetching.

4 EXPERIMENTS AND EVALUATION

4.1 Experiment Setup

Considering SSD controller has limited computation power and memory capacity, we conducted tests on a local ARM-based machine that has an ARM Cortex A7 Dual-Core with 800 MHz, 128 MB of memory, and 32-bit Linux (ver 3.1). We have performed trace-driven simulation with *SSDsim (ver2.1)* on the local machine, which has a diverse set of configurations and supports of *TLC* flash simulation [33, 35]. We have integrated our proposal with *SSDsim*, for supporting data prefetching inside of SSDs. Table 1 presents the default settings of *SSDsim* in our experiments, which were decided by either referring to prior studies [33, 34], or carrying out sensitive tests. Specifically, *Cache size* is 32 MB by default, which is smaller than 0.1% of a 128 GB SSD device [36]. The relevant sensitive analysis on scalability with varying size of cache will be described in Section 4.5.2.

The metric of *Matching hit threshold* in the table implies the condition of triggering a prefetching operation after pattern matching, and the default value of *matching hit threshold* is configured as 50%, and the related sensitive analysis will be found in Section 4.5.1. The number of I/O requests in each time window is setting to 1,024, and the first 256 requests in the window are used to disclose frequent patterns to guide data prefetching. In addition, to reflect the impact of garbage collections, before running traces, the simulated SSD is aged to that with 70% of its capacity being used [33, 34].

We employed nine commonly used disk traces [28, 29]. Specifically, *websearch_1* and *websearch_2* (label as *web1* and *web2*) are collected from UMass Trace Repository [30], and the next five traces are from the block I/O trace collection of Microsoft Research Cambridge [31]. The remaining two recent block I/O traces are recently collected from a part of an enterprise **virtual**

Table 1. Experimental Settings of *SSDsim* (TLC Cell)

Parameters	Values	Parameters	Values
Capacity	128 GB	*Read time*	0.075 ms
Page per block	64	*Write time*	2 ms
Page size	8K	*Cache read/write*	0.001 ms
Cache size	32 MB	*Elements in pattern*	[2, 10]
Fixed prefetch cache	8%	*Matching hit threshold*	50%

Note: α and β in Equation (7) are then `0.074 ms` and `1.999 ms`.

ALGORITHM 1: Adaptive cache replacement policy

```
Input: args of $read\_addr\_space$, $\theta\_cur$, $\theta\_prev$;
Output: completion of I/O processing in a time window;
/* quantifying the number of pages for prefetched data */
$rd\_pages\_cur = read\_addr\_space \times \theta\_cur$;
$rd\_pages\_prev = read\_addr\_space \times \theta\_prev$;
/*0: replacing read pages, 1: replacing write pages*/
$rep\_flag = 0$;
/* processing requests in the current time window */
for req in I/O Queue do
    if $req \rightarrow addr$ hit in Cache then
        continue;
    end
    $rep\_flag = rd\_pages\_cur \geq rd\_pages\_prev$ ? 1:0;
    /*evicting cached data and buffering new written data*/
    if req is a Write then
        lru_replace($req \rightarrow size, rep\_flag$)
        if !$rep\_flag$ then
            $rd\_pages\_cur$ -= $req \rightarrow size$;
        end
        load_in_cache($req \rightarrow data$);
    end
    /*carrying out pattern-based data prefetching */
    else if $req \rightarrow addr$ hit in $read\_pattern$ then
        pattern_prefetch(&$buf$, $read\_pattern$);
        lru_replace($buf \rightarrow size, rep\_flag$);
        if $rep\_flag$ then
            $rd\_pages\_cur$ += $buf \rightarrow size$;
        end
        load_in_cache($buf \rightarrow data$);
    end
end
/* preparing the ratio parameter for next round. */
$\theta\_prev = \theta\_cur$;
```

desktop infrastructure (**VDI**) [32]. Specifically, they are additional-05-1908-LUN1 (lun1) and additional-05-1815-LUN1 (lun2). Among the selected block I/O traces, the evaluated workloads

Table 2. Specifications on Selected Disk Traces (in Reverse Order By Read Ratio)

Trace	# of Req.	Read ratio	Avg. read size	Hot read	Read/total footprint
*web*1	1,055,448	99.9%	15.1 KB	94.2%	0.02/0.02 GB
*web*2	4,579,809	99.9%	15.0 KB	98.4%	0.02/0.02 GB
*hm*1	609,311	95.3%	14.9 KB	42.0%	0.15/0.20 GB
*lun*1	1,249,625	66.8%	22.2 KB	1.1%	13.15/15.21 GB
*lun*2	1,370,658	59.8%	22.6 KB	2.4%	12.74/16.97 GB
*usr*0	2,237,889	40.5%	40.9 KB	49.4%	2.09/2.44 GB
*hm*0	3,993,316	35.5%	7.4 KB	32.7%	1.85/2.31 GB
*src*1_2	1,907,773	25.4%	19.1 KB	56.0%	1.56/1.97 GB
*wdev*0	1,143,261	20.1%	12.6 KB	69.3%	0.20/0.52 GB

are composed of three read-dominated traces, three read-write-balanced ones, and three write-intensive ones. The detailed specifications on the traces are reported in Table 2. The metric of *hot read* in the table represents the ratio of hot access addresses if they have been requested not less than three times.

The following schemes are used in evaluation tests, including the baseline without prefetching and other prefetching approaches.

- *Baseline*: which indicates the mechanism of data prefetching is not supported. That is to say, the SSD cache is only used for buffering written data.
- *Baseline-RB*: which utilizes the SSD DRAM for caching both write and read data, while the SSD DRAM of *Baseline* is only used for write data.
- *C-Miner*: which is a prefetching method by exploiting mining block correlations in storage systems [27]. To fairly compare with other methods, this approach is implemented inside of SSDs (not at the file system level), and adopts a fixed size of prefetch cache.
- ***Resource-Optimized Prefetching (ROP):*** which makes use of a Markov chains-based learning algorithm to predict batches of future reads, for eventually directing data prefetching [20]. We argue that *ROP* is one of the works most related to ours, as it is implemented inside of SSDs and has the features of OS dependence [20].
- *Pattern*: which is our previous work on SSD-inside prefetching [35]. It does support pattern-based prefetching and makes use of an empirical rule to divide the SSD cache, for separately buffering the written data and the prefetched data.
- *Pattern+*: which is the newly proposed prefetching scheme in the article. It is able to more precisely support adaptive cache partition, by using the proposed comprehensive model.

4.2 Tests and Benefit Illustration

To measure the validity of the proposed prefetching mechanism that aims to enhance the prefetching accuracy and the cache use efficiency, we leverage the following three metrics in our tests: (a) *I/O response time*, (b) *prefetching hit per prefetch*, and (c) *cache use efficiency*.

4.2.1 Read Response Time. Figure 7 shows the average read latency after replaying the selected block traces by using varied prefetching schemes and two *Baseline* methods. As seen, four prefetching schemes can achieve an improvement on the read response time, compared with *Baseline* that does not support prefetching. Intuitively, data prefetching can yield attractive performance improvements in the traces that have a high read ratio of requests (e.g., *web1, web2, hm1*) and a relative large value of *hot read* (e.g., *usr0, src1_2, wdev0*). The noticeable clue shown in the figure is that *Baseline-RB* achieves the best read performance when replaying the traces of *web1* and *web2*,

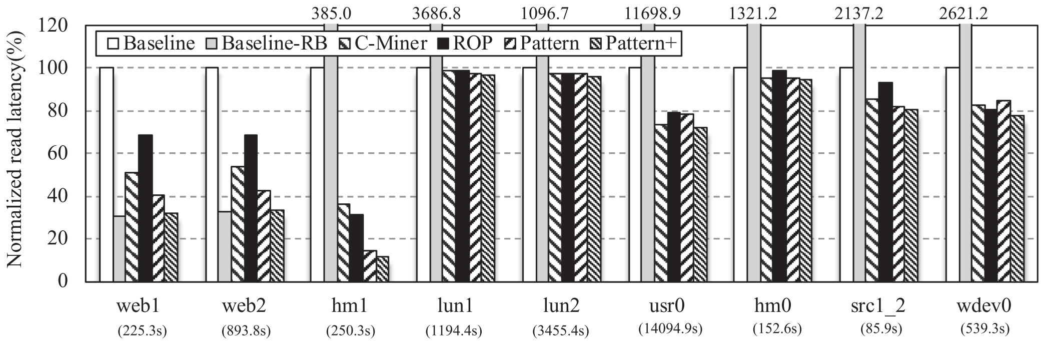

Fig. 7. Read latency of running selected block traces. Note that the absolute values using *Baseline* are shown below the trace names.

but the worst read performance when replaying other selected traces. On the one hand, both web traces have few write requests (there is even no GC operations after running both of them; see Section 4.3.1), so that almost all the SSD cache can be devoted for buffering hot read data that contribute to better read performance. On the other hand, *Baseline-RB* works very poorly when running other traces having more write requests. This is because buffering the read data decreases the cache space for the write data and then triggers more GC operations, which will delay the routine processing on read/write requests.

Among all prefetching schemes, the proposed *Pattern+* scheme can reduce the read latency by 33.8%, 8.8%, and 13.5% on average, in contrast to *Baseline*, *C-Miner*, and *ROP*. This is because *Pattern+* can more accurately prefetch data, which are most possibly accessed by the following read requests by referring to the mined frequent access patterns. More importantly, *Pattern+* can further cut down the read response time by 4.3% on average, compared with *Pattern*. This is because *Pattern* employs a simple empirical rule to determine the cache partition, but *Pattern+* does have a theoretical model for accurately guiding cache partition while running the benchmarks. Then, *Pattern+* is able to better make use of the cache and has a shorter read response time.

It is worthy to mention that our proposed method of *Pattern+* has a slight improvement on read latency compared with *Baseline* in dealing with the trace of *hm0*, though prefetching in *Pattern+* only works in the first three time windows. We think this is because the fast processing on I/O requests in the first few time windows will reduce the number of I/O requests in the waiting queue in the following windows, and the prefetched data are gradually ejected from the prefetch cache that also benefit read requests if their expected data are still in the prefetch cache.

4.2.2 Write and Overall I/O Time. Buffering the prefetched data in SSD cache must place negative effects to the performance of writing data, since a part of SSD cache is separated for the prefetched data. The result of write response time is shown in Figure 8. Clearly, all prefetching approaches slightly increase the write response time in the write-intensive workloads compared with *Baseline*. The proposed *Pattern+* scheme works the best among prefetching approaches and it only increases the write response time by 0.22% on average. In the worst case of running the trace of *hm1* by comparing with *Baseline*, the prefetching schemes of *C-Miner*, *ROP*, *Pattern*, and *Pattern+* result in more write latency by 6.3%, 6.5%, 7.6%, and 4.4%, respectively.

We then record the overall I/O time to show the effectiveness of data prefetching after replaying the selected traces, and Figure 9 presents the results. As seen, the proposed method of *Pattern+* performs the best, and can cut down the total I/O latency by 18.5%, 5.1%, 9.0%, and 3.3% on average, in contrast to *Baseline*, *C-Miner*, *ROP*, and *Pattern*. On the other hand, we can understand

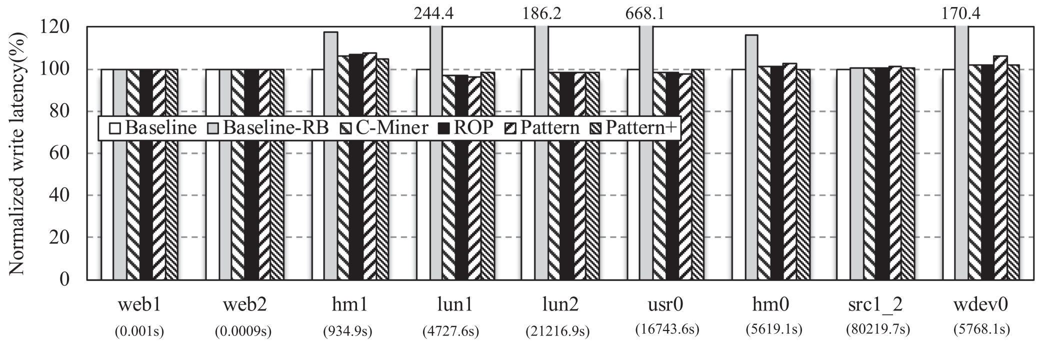

Fig. 8. Write latency of running selected block traces.

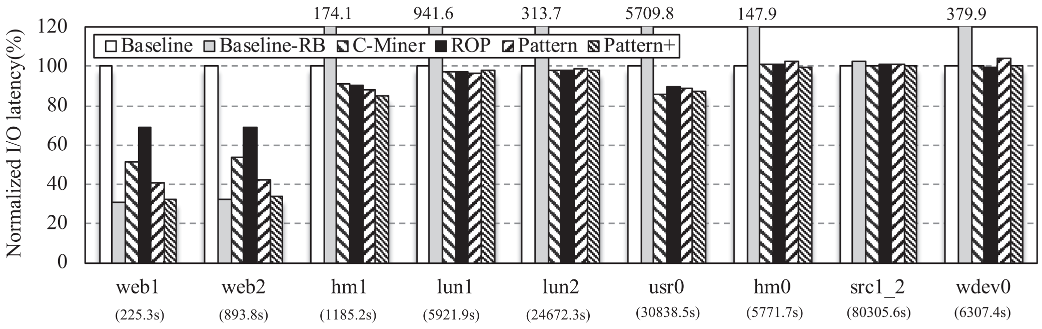

Fig. 9. The total processing time of running selected block traces.

that data prefetching does not make sense on the reduction in overall I/O time while processing *hm0*, *src1_2*, and *wdev0*. This is because these block I/O traces have a relative high ratio of write requests, aggressively prefetching fails to offset the overhead of moving certain written data out of the cache.

More importantly, *Pattern+* can dynamically minimize or even remove the prefetch cache, when running the write-intensive workloads or prefetching does not enhance I/O performance. Then, *Pattern+* outperforms other prefetching schemes (i.e., *ROP* and *Pattern*), and yields a similar overall I/O time to *Baseline* while replaying the write-intensive I/O traces. In brief, *Pattern+* can achieve the best overall I/O time if data prefetching works well for the applications, but it does not bring about many negative impacts on total I/O time in case data prefetching does not work.

4.2.3 Prefetching Hit Per Prefetch. To measure the effectiveness of different prefetching schemes, we record the 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 reads on the prefetched data in the unit of page, and the prefetch count means the number of prefetched pages. In the case of the same size of the prefetch cache, the larger *prefetching hits per prefetch* indicates better prefetching effectiveness.

Figure 10 shows the results of *prefetching hits per prefetch* by using three prefetching approaches. Since *Baseline* does not conduct data prefetching, it is not included in the figure. As seen, *Pattern+* can noticeably enhance the metric of *prefetching hits per prefetch* by `1.44x` and `3.67x` on average compared with *C-Miner* and *ROP*. Another interesting clue shown in the figure is that *Pattern+* shows lower *prefetching hit per prefetch*s than *Pattern*. This is because *Pattern+* commonly mini-

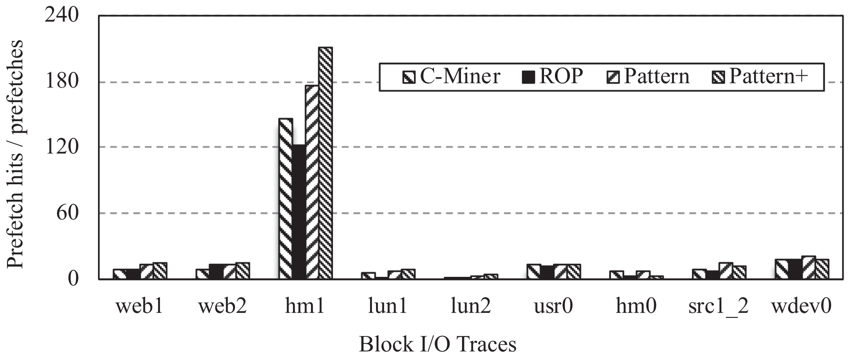

Fig. 10. The results of prefetching hits per prefetch.

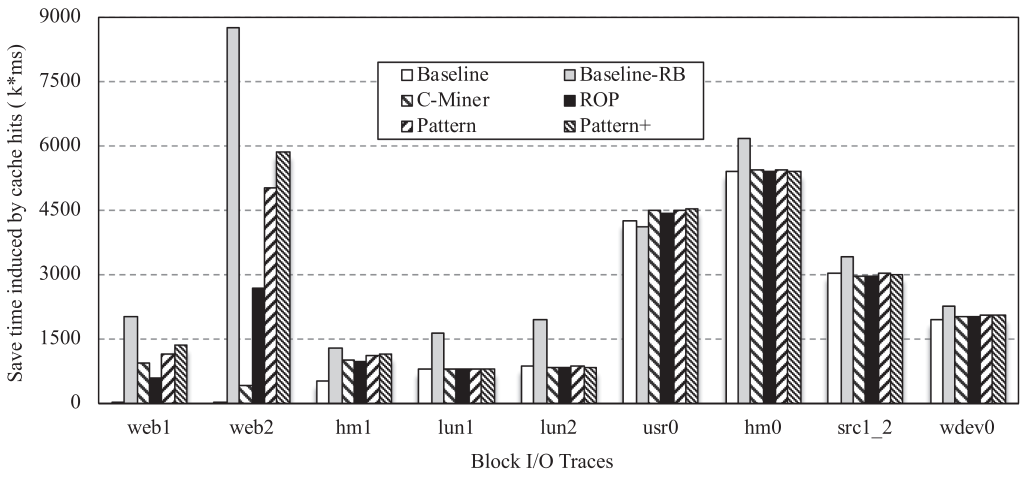

Fig. 11. Cache use efficiency: the total saved time induced by cache hits.

mizes and even removes the prefetch cache to `0` in some periods of (write-intensive) workloads. Consequently, it may bring about less prefetching operations in such contexts and then has a lower *prefetching hits per prefetch* in contrast to *Pattern*. But, *Pattern+* is able to make more cache space for the written data and to ensure a better overall I/O response time, as discussed in Section 4.2.2.

4.2.4 Cache Use Efficiency. We define the indicator of *cache use efficiency* as the saved time induced by cache hits. It is the sum of the total read cache hits multiplied by α and the total write cache hits multiplied by β, by referring back to Equation (7).

As seen in Figure 11, *Baseline* results in the lowest values of the saved time by cache hits when replaying read-intensive traces. This is because prefetching does work well for such applications, but *Baseline* does not support it. *Baseline-RB* shows the highest value of *cache use efficiency* in the most cases, but it poses threats to write cache, and thus induces more write data to be ejected from the write cache. In addition, we see the proposed *Pattern+* method can also improve the metric of *cache use efficiency* by `1.29x` and `1.04x`, compared with the related work of *ROP* and *Pattern*. *Pattern+* adaptively splits the cache into the prefetch cache and the write cache, and then achieves the best *cache use efficiency*.

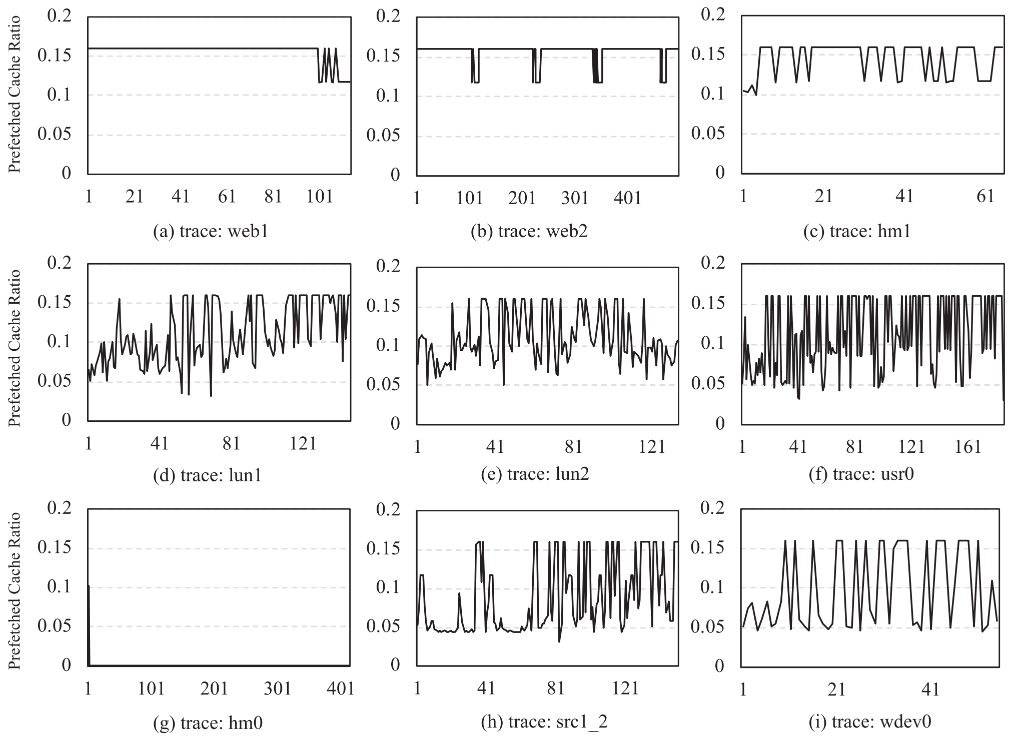

Fig. 12. Ratio of prefetch cache to all cache.

4.2.5 Analysis on Adaptive Cache Partition. The related approaches of *ROP* and *C-Miner* employ a fixed 8% cache for buffering the prefetched data [20]. On the other hand, adaptive cache partition aims to allot an appropriate part of cache as the prefetch cache by referring the characteristics of I/O workloads and the historical prefetching productivity. In our design, it turns off the prefetching functionality if prefetching may not bring about positive effects; that means the portion of prefetch cache becomes 0, and all cache spaces are devoted for written data.

Considering applications may have varied read/write states in the different stages, we then set an upper limit (double of the default 8% in comparison counterparts) to avoid extreme cases. To be specific, we think a large upper limit may worsen write performance in some cases. For instance, the application has intensive write workloads in the forthcoming time window but had intensive reads and prefetching that worked very well in the previous time window. Figure 12 presents the statistics on the ratio of prefetch cache to all cache. As shown, it adopts different ratios for different traces, and even varied ratios in different time windows for the same trace.

Because the traces of *web1, web2, and hm1* are read-heavy, the proposed scheme of adaptive cache management reaches the upper limit of 16% in the most stages of workloads. Specifically, we see prefetching is disabled in most time windows in the case of *hm0*, as prefetching will pose negative impacts on write performance. Regarding other block traces, *Pattern+* adopts the varied size of prefetch cache in different time windows, for adaptively making use of whole SSD cache to keep the prefetched data and the written data. In brief, adaptive cache management can contribute to read performance improvements and more efficient cache use management, whenever running the workloads with varied characteristics.

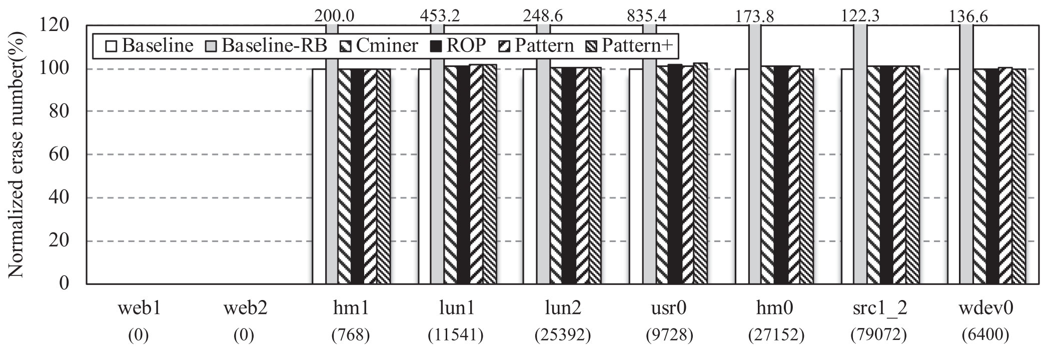

Fig. 13. Erase statistics caused by garbage collection.

4.3 Overhead Analysis

This section presents the overhead caused by our proposed mechanism. It first analyzes the garbage collection erases caused by write amplification. Then, it analyzes the time consumption caused by pattern mining and pattern matching. After that, the space overhead resulted by holding frequent access patterns and the matching matrix will be reported.

4.3.1 Erase Overhead. Figure 13 indicates the erase number comparison after running all workloads. The results are normalized to those of *Baseline*, and the absolute erase numbers using *Baseline* are shown below the trace names. Two read-intensive workloads (i.e., *web1 and web2*) do not have any GC operations since the available space after running them is not less than the predefined GC threshold; meanwhile, the other workloads have different rates of normal GC erases. As seen, *Baseline-RB* shows more erase numbers more by 119%, in contrast to *baseline*. This is because *Baseline-RB* buffers both read and write data in the SSD DRAM, and then a large number of write data are ejected onto SSD chips, which must lead to more garbage collections. In fact, this is the reason why *Baseline-RB* achieves the best I/O response time in the read-dominant traces, but the worst I/O response time in other selected traces having GC operations.

On the other hand, we can understand that all prefetching methods result in more erase operations, since separating a part of SSD DRAM for holding the prefetched data must eject more write data onto SSD chips (i.e., write amplification). More exactly, *ROP*, *C-Miner*, *Pattern*, and *Pattern+* result in more erase operations by 0.99%, 1.00%, 0.92%, and 0.83% on average, compared with *Baseline*.

4.3.2 Time Overhead. The comparison counterparts of *Baseline* and *ROP* do not carry out mining frequent patterns and matching relevant current requests with mined patterns, so that they do not bring about any mining and matching overhead. But, our pattern-based prefetching scheme does result in certain mining and matching overhead. In general, the mining and matching overhead is related to the number of total requests and the number of frequently read addresses in the trace. Figure 14 shows the time required by mining hot read pattern and matching the current requests with the mined patterns, when using *Pattern* and *Pattern+*. In the figure, the bars represent the time overhead, and the dots indicate the ratio of such overhead to total I/O processing time.

As seen, the mining and mapping overhead is less than 4.46 seconds for all traces and less than 1.39% of the total I/O processing time, which is acceptable. In the traces except for read-intensive ones (i.e., except for *web1, web2*), it only takes less than 0.068% of the total I/O processing time. Note that the results of read/overall I/O time reported in Sections 4.2.1 and 4.2.2 have considered the impact of time overhead toward I/O response time.

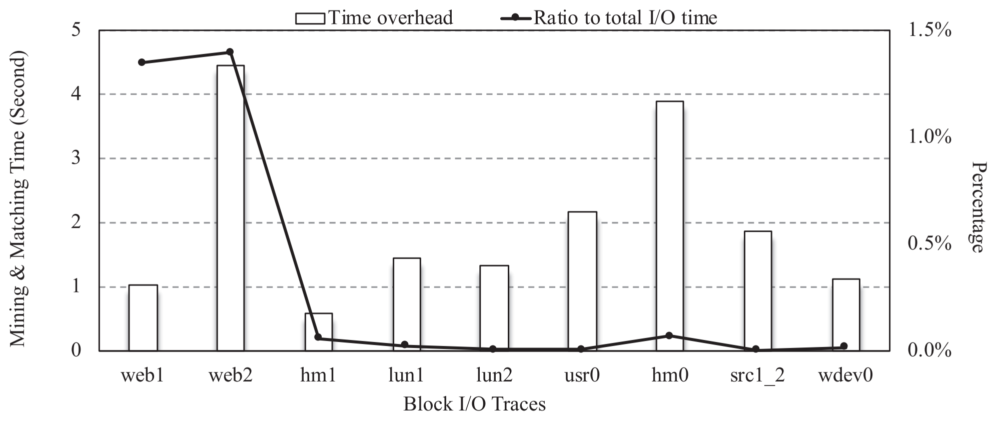

Fig. 14. Frequent pattern mining and matching overhead in *Pattern* and *Pattern+*.

Table 3. The Mined Pattern Statistics on Selected Traces

Trace	Avg. pattern size	Trace	Avg. pattern size
*web*1	5.6	*usr0*	6.5
web2	5.5	*hm0*	5.4
*hm*1	4.9	*src1_2*	6.3
*lun*1	3.0	*wdev0*	6.5
*lun*2	7.3		

4.3.3 Space Overhead. The space overhead is mainly caused by holding mined patterns and the match matrix, and Table 3 shows the statisitical data on mined frequent access patterns in the selected traces. For example, each pattern consists of 6.5 logical sector addresses on average after running *usr0*.

Our proposal does require extra memory space for holding the information to direct pattern-based prefetching, but it can improve the prefetching efficiency and thus reduce I/O latency. In fact, the matching matrix is updated in units of time windows and each time window has 1,024 requests by default. We observe that only a very small part of windows have more than 32 mined patterns and 128 logical addresses in all mined patterns, which is 2.5% and 3.7% on average after running all selected traces. Therefore, we construct and maintain the matching matrix having 128 columns and 32 rows in our tests, for not wasting memory space to hold the matching matrix. In the worst case, it holds a matching matrix to maintain the pattern information and to carry out the pattern matching, which only consumes 5.625 KB (=32 (rows)*128 (columns)*1bit (recording 0 or 1) +32 (rows)*4B (recording an address number) +128 (columns)*10 (maximum pattern element) *4B) taking a negligible amount of memory space in SSDs. Specifically, 1bit means each record content (i.e., 0 or 1) in the matching matrix, and 4B represents space overheads for holding each entry information of both row and column.

4.4 Case Study on Hybrid Workloads

Write amplification aggravates garbage collection operations and then affects I/O performance. In order to verify the effectiveness of our proposal for running complicated and long-time workloads, this section conducts a case study to run large hybrid workloads. We compose seven hybrid workloads by combining the selected traces previously reported in Table 2, and the detailed combination

Table 4. Specifications on hybrid traces

Trace	Components	Read/total volume	Read/total footprint
whole	9 selected traces	181.42/284.08 GB	26.95/34.03 GB
*read*3	3 read-domained traces	89.33/89.88 GB	0.17/0.21 GB
*balance*3	3 balanced traces	70.56/100.90 GB	25.09/31.54 GB
*write*3	3 write-intensive traces	21.53/93.31 GB	2.39/3.57 GB
*hybrid*1	read3 + balance3	159.89/190.78 GB	25.19/31.67 GB
*hybrid*2	read3 + write3	110.86/183.18 GB	2.46/3.63 GB
*hybrid*3	balance3 + write3	92.10/194.20 GB	26.91/33.99 GB

specifications are presented in Table 4. Figure 15 shows the results of critical performance metrics (i.e., I/O response time and erase numbers), after running the hybrid block I/O traces.

As seen in Figure 15(a), *Pattern+* yields the least read latency, which further proves the effectiveness of the proposed prefetching approach with adaptive cache adjustment. On the other hand, all prefetching schemes result in a slight drop in write performance, as demonstrated in Figure 15(b). Specifically, *Pattern+* leads to more write time by merely 0.51% on average, compared with *Baseline*. This is due to the fact that separating a part of SSD DRAM as a prefetch cache will cause write amplification, as discussed in Section 4.2.2. When it comes to the overall I/O performance shown in Figure 15(c), *Pattern+* decreases the overall I/O performance by 8.71%, 3.24%, 5.09%, and 1.46%, compared with *Baseline*, *C-Miner*, *ROP*, and *Pattern*.

With respect to the impacts of write amplification when running large workloads, we count the number of erase operations induced by garbage collections, and Figure 15(d) presents the results. As shown, the proposed method of *Pattern+* only causes more garbage collections by less than 1.23%, compared with *Baseline*. Then, we conclude that our proposal can also work well for long-time workloads, and does not trigger too much garbage collection operations to impact the performance of the I/O operations.

4.5 Sensitivity and Scalability

4.5.1 Sensitivity Analysis. Because the elements of frequent access patterns rarely appears more than 10, we set the pattern elements between 2 and 10. This section presents the sensitive study on the pattern *matching hit threshold*. Because the minimum pattern element is 2, the sensitive study *matching hit threshold* starts from 50%. Figure 16 shows the results of *prefetching hits per prefetch* (i.e., the prediction accuracy) and the normalized I/O response time by employing varied thresholds in pattern matching. As seen in Figure 16(a), the measure of *prefetching hits per prefetch* slightly improves when the *matching hit threshold* is becoming larger. But note that a higher value of *matching hit threshold* indicates less prefetching operations even though they have similar tendency on *prefetching hits per prefetch*. Let us take *wdev0* as an example; we count the prefetching operations under the threshold of 90% are 4,272 times less than that under 50%. As a result, the improvements on read latency caused by prefetching become different, which have been illustrated in Figure 16(b). In brief, by considering both measurements of prefetching accuracy and I/O performance, we select 50% as the default one in our tests.

In order to determine the size of time window in our evaluation, we set different values of 256, 512, 1,024, and 2,048 to carry out sensitive analysis. Figure 17 shows the results of the *prefetching hits per prefetch* (i.e., the prediction accuracy) and normalized I/O response time by adopting different time window sizes. As seen in Figure 17(a), 1,024 and 2,048 of time window size yield good prediction accuracy in selected block I/O traces. This is because more history information can result in a higher prediction accuracy. Moreover, we see that 1,024 of time window size unveils

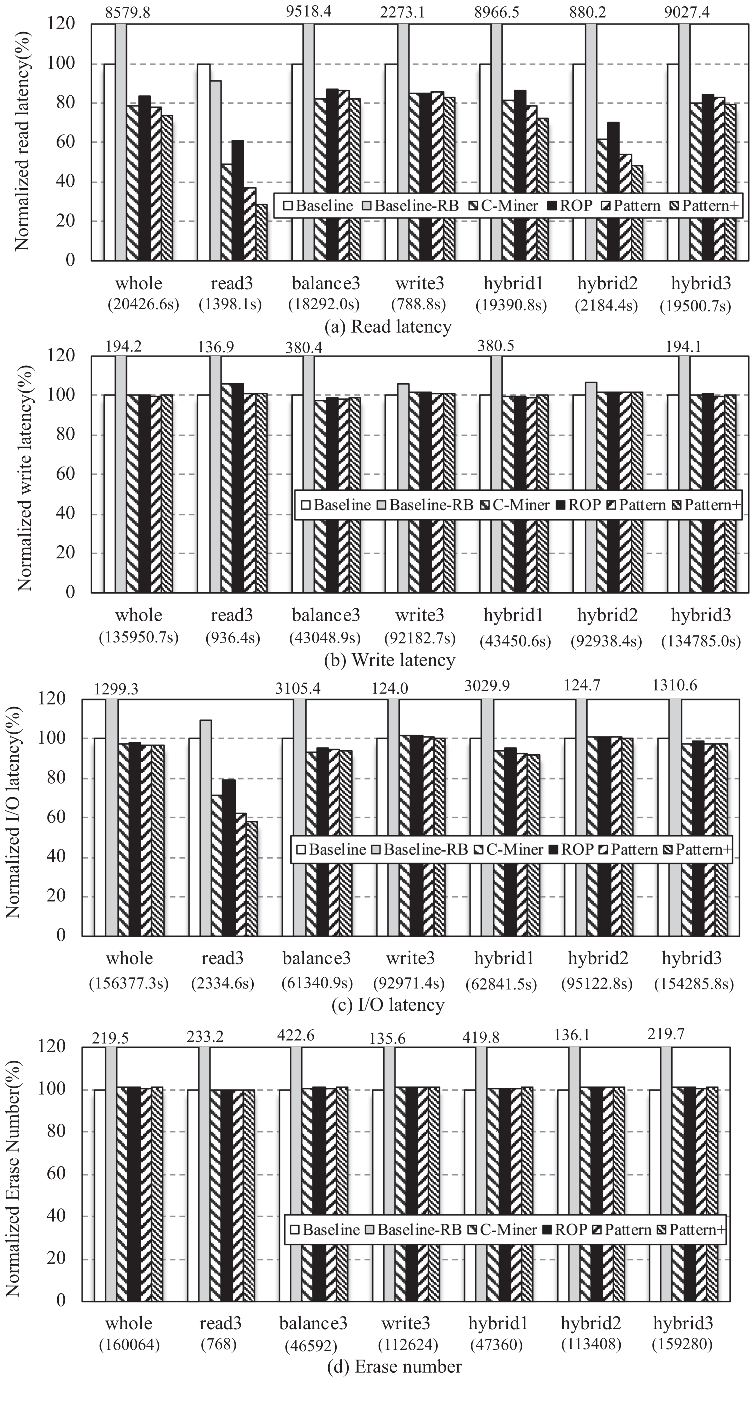

Fig. 15. I/O performance and erase on hybrid workloads.

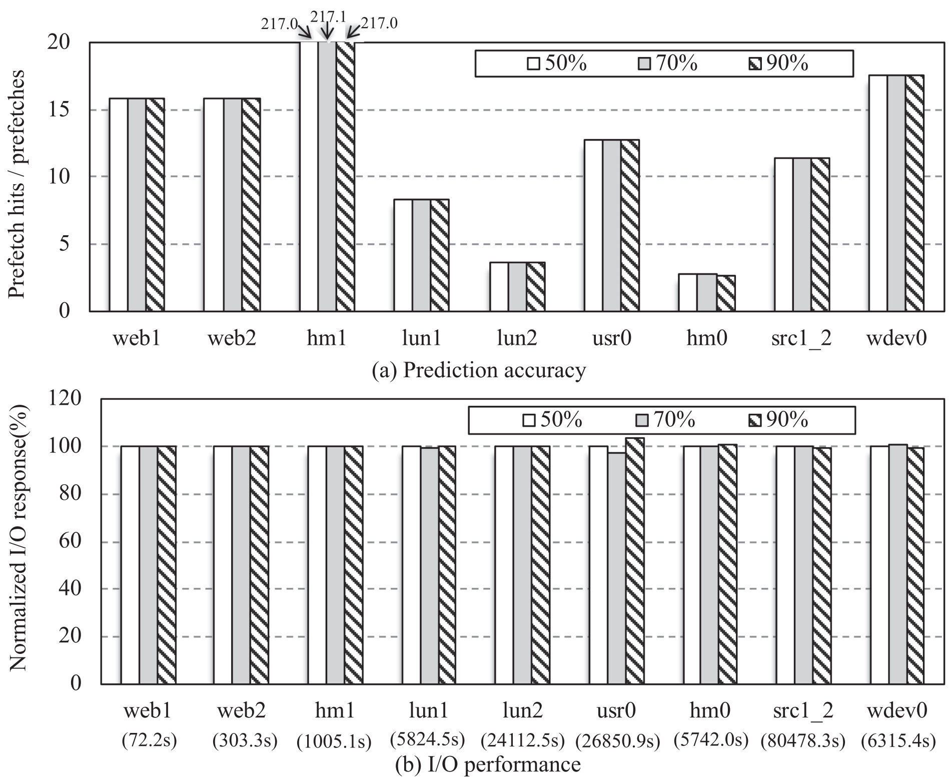

Fig. 16. Sensitive analysis on pattern matching hit ratio.

the best I/O performance in the most traces, as demonstrated in Figure 17(b). Then, by considering both the prefetching accuracy and the I/O response time, we set `1,024` as the default time window size in our tests.

4.5.2 Scalability Analysis. This section reports the scalability of *Pattern+* on varied sizes of SSD cache, and Figure 18 discloses the detailed results about different cache sizes from `4M` to `64M`. Our proposed schemes of *Pattern* and *Pattern+* achieve a noticeable improvement on I/O response time compared with *Baseline* and *ROP*. This fact further verifies that our proposal can yield a good scalability.

Specifically, in the cases of running *web1* and *web2*, *Baseline* does not conduct any prefetching scheme, so it cannot achieve I/O improvement in these two read-intensive traces, even though the cache size does increase. In addition, when the cache size increases, all schemes do not yield obvious improvements on I/O response time while replaying the traces of *lun1* and *lun2*. This is because these two traces have few hot access addresses (i.e., the number of cache hits is confined), and then a large cache size cannot notably benefit I/O performance improvements.

4.6 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,

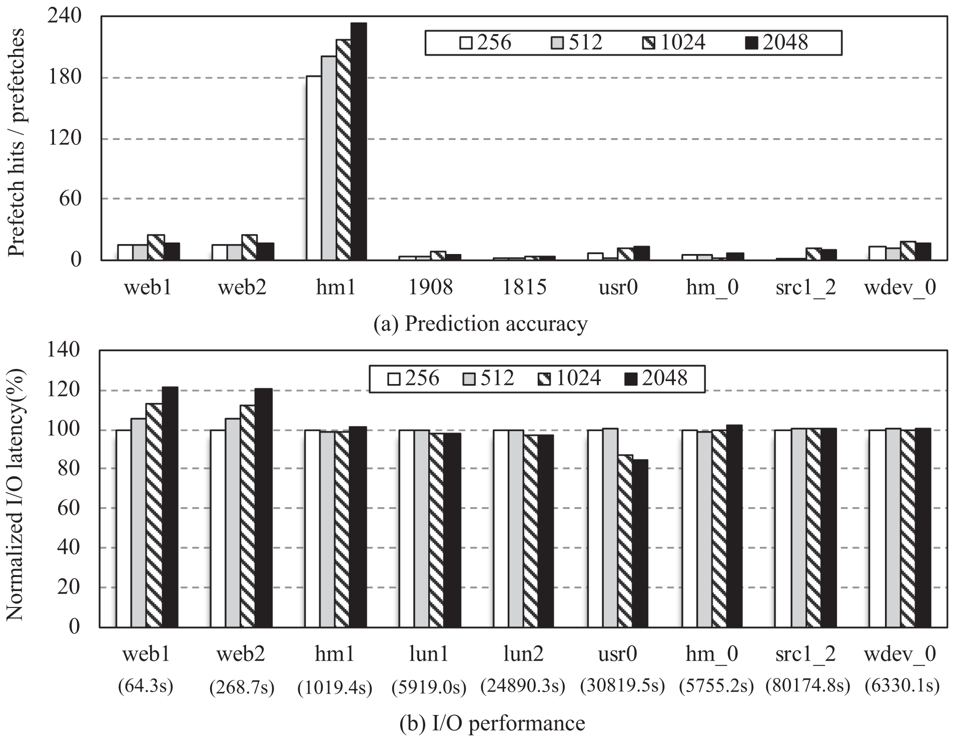

Fig. 17. Sensitive analysis on time window size.

by avoiding unnecessary prefetching operations. *Second*, the proposed scheme of adaptive cache management, which is based on the theoretical model, can dynamically divide the SSD cache into the prefetch cache and the write cache, to further boost the efficiency of cache use. In brief, we conclude that the proposed prefetching scheme is able to significantly enhance the effectiveness of data prefetching and yield a better I/O performance improvement.

5 CONCLUSIONS

This article 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. More importantly, in order to maximize the use efficiency of SSD cache, we have built a mathematical model for supporting adaptive cache management. It dynamically separates SSD cache into the write cache and prefetch cache, for respectively buffering the written data and prefetched data, on the basis of the I/O characteristic of workload and the prefetching efficiency. In brief, our proposed prefetching scheme can better take advantage of SSD cache and thus improve the I/O performance.

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 36.5% on average. In addition, 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 2.2x, in contrast to other comparison counterparts.

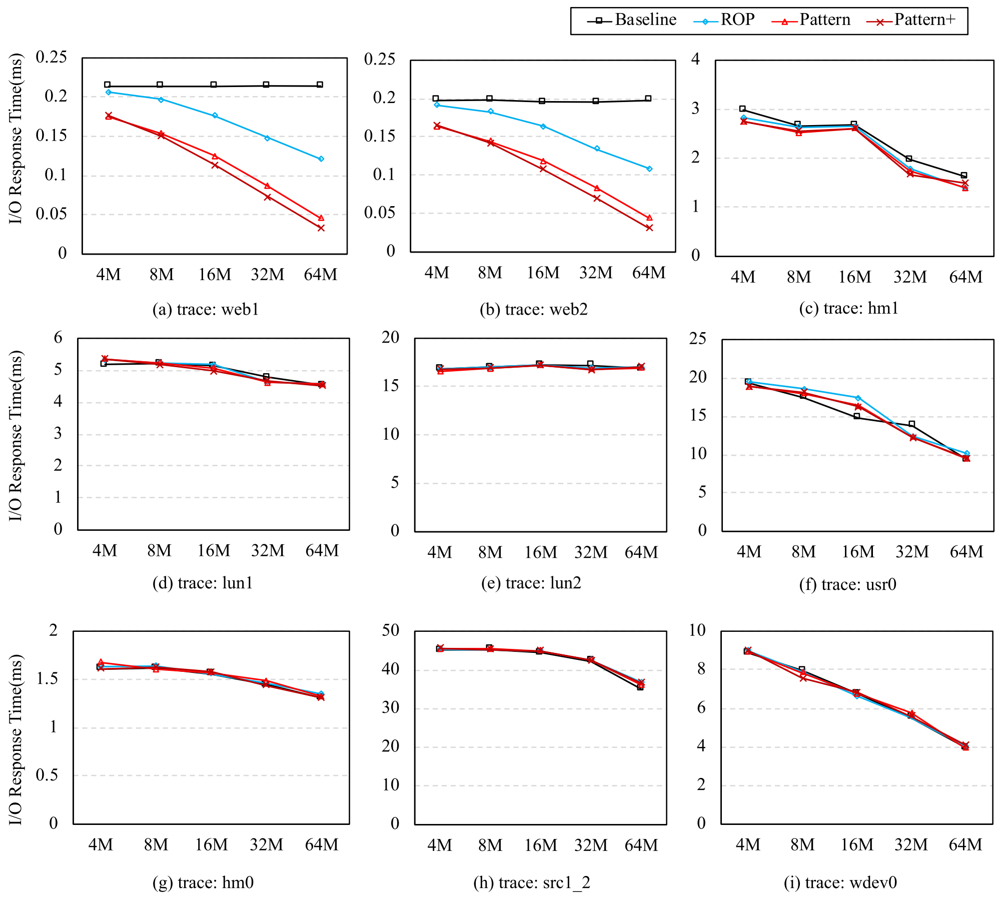

Fig. 18. Scalability analysis on cache sizes.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers of *DATE'20* for their thorough reviews and insightful comments.

REFERENCES

[1] Laura M. Grupp, John D. Davis, and Steven Swanson. 2012. The bleak future of NAND flash memory. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'12).*

[2] Rino Micheloni. 2017. Solid-State Drive (SSD): A nonvolatile storage system. In *Proceedings of the IEEE* 105, 4 (2017), 583–588. DOI: https://doi.org/10.1109/JPROC.2017.2678018

[3] Chihiro Matsui, Chao Sun, and Ken Takeuchi. 2017. Design of hybrid SSDs with storage class memory and NAND flash memory. In *Proceedings of the IEEE* 105, 9 (2017), 1812–1821. DOI: https://doi.org/10.1109/JPROC.2017.2716958

[4] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo Kim. 2008. A reconfigurable FTL (flash translation layer) architecture for NAND flash-based applications. In *ACM Transactions on Embedded Computing Systems* 7, 4 (2008), 38:1–38:23. DOI: https://doi.org/10.1145/1376804.1376806

[5] Yuan-Hao Chang, Po-Liang Wu, Tei-Wei Kuo, and Shih-Hao Hung. 2012. An adaptive file-system-oriented FTL mechanism for flash-memory storage systems. In *ACM Transactions on Embedded Computing Systems* 11, 1 (2012), 9:1–9:19. DOI: https://doi.org/10.1145/2146417.2146426

[6] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. 2011. Leveraging value locality in optimizing NAND flash-based SSDs. In *Proceedings of USENIX Conference on File and Storage Technologies (FAST'11).*

[7] Jianwei Liao, Fengxiang Zhang, Li Li, and Guoqiang Xiao. 2015. Adaptive wear-leveling in flash-based memory. In *IEEE Computer Architecture Letters* 14, 1 (2014) 1–4.
[8] Yili Wang, Kyung Tae Kim, Byung Jun Lee, and Hee Yong Youn. 2018. A novel buffer management scheme based on particle swarm optimization for SSD. In *The Journal of Supercomputing*. 74, 1 (2018), 141–159.
[9] Jinhua Cui, Youtao Zhang, Jianhang Huang, Weiguo Wu, and Jun Yang. 2018. ShadowGC: Cooperative garbage collection with multi-level buffer for performance improvement in NAND flash-based SSDs. In *Proceeding of Design, Automation & Test in Europe Conference & Exhibition (DATE'18)*. DOI: https://doi.org/10.23919/DATE.2018.8342206
[10] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. 2009. Improving flash wear-leveling by proactively moving static data. In *IEEE Transactions on Computers* 59, 1 (2009), 53–65. DOI: https://doi.org/10.1109/TC.2009.134
[11] Jun Li, Xiaofei Xu, Xiaoning Peng, and Jianwei Liao. 2019. Pattern-based write scheduling and read balance-oriented wear-leveling for solid state drivers. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST'19)*. DOI: https://doi.org/10.1109/MSST.2019.00-10
[12] Elizabeth A. M. Shriver, Christopher Small, and Keith A. Smith. 1999. Why does file system prefetching work?. In *Proceedings of the USENIX Annual Technical Conference (ATC'99)*.
[13] Ahsen J. Uppal, Ron Chi-Lung Chiang, and H. Howie Huang. 2012. Flashy prefetching for high-performance flash drives. In *Proceedings of the Symposium on Mass Storage Systems and Technologies (MSST'12)*. DOI: https://doi.org/10.1109/MSST.2012.6232367
[14] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. 2016. Lynx: A learning linux prefetching mechanism for SSD performance model. In *Proceedings of the Non-Volatile Memory Systems and Applications Symposium (NVMSA'16)*. DOI: https://doi.org/10.1109/NVMSA.2016.7547186
[15] Cosmos OpenSSD Platform. Retrieved September 2020 from http://www.openssd-project.org.
[16] Song Jiang, Xiaoning Ding, Yuehai Xu, and Kei Davis. 2013. A prefetching scheme exploiting both data layout and access history on disk. In *ACM Transactions on Storage* 9, 3 (2013), 10:1–10:23. DOI: https://doi.org/10.1145/2508010
[17] Jianwei Liao, François Trahay, Balazs Gerofi, and Yutaka Ishikawa. 2016. Prefetching on storage servers through mining access patterns on blocks. In *IEEE Transactions on Parallel and Distributed Systems* 27, 9 (2016), 2698–2710. DOI: https://doi.org/10.1109/TPDS.2015.2496595
[18] Jun He, John Bent, Aaron Torres, Gary Grider, Garth Gibson, Carlos Maltzahn, and Xian-He Sun. 2013. I/O acceleration with pattern detection. In *Proceedings of the International Symposium on High-Performance Parallel and Distributed Computing (HPDC'13)*. DOI: https://doi.org/10.1145/2462902.2462909
[19] Leeor Peled, Uri C. Weiser, and Yoav Etsion. 2018. Towards memory prefetching with neural networks: Challenges and insights. https://arxiv.org/pdf/1804.00478v1.pdf.
[20] Rui Xu, Xi Jin, Linfeng Tao, Shuaizhi Guo, Zikun Xiang, and Teng Tian. 2018. An efficient resource-optimized learning prefetcher for solid state drives. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'18)*. DOI: https://doi.org/10.1145/2462902.2462909
[21] Devesh Tiwari, Simona Boboila, Sudharshan S. Vazhkudai, Youngjae Kim, Xiaosong Ma, Peter Desnoyers, and Yan Solihin. 2013. Active flash: Towards energy-efficient, in-situ data analytics on extreme-scale machines. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'13)*.
[22] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: Creating application objects efficiently for heterogeneous computing. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'16)*. DOI: https://doi.org/10.1109/ISCA.2016.15
[23] Sang Woo Jun, Ming Liu, Sungjin Lee, Jamey Hicks, John Ankcorn, Myron King, Shuotao Xu, and Arvind. 2015. BlueDBM: An appliance for big data analytics. In *Proceedings of the ACM/IEEE Annual International Symposium on Computer Architecture (ISCA'15)*. DOI: https://doi.org/10.1145/2749469.2750412
[24] Shuyi Pei, Jing Yang, and Qing Yang. 2019. REGISTOR: A platform for unstructured data processing inside SSD storage. In *ACM Transactions on Storage* 15, 1 (2019), 7:1–7:24. DOI: https://doi.org/10.1145/3310149
[25] Richard A. Groeneveld and Glen Meeden. 1984. Measuring skewness and kurtosis. In *Journal of the Royal Statistical Society: Series D (The Statistician)*. DOI: https://doi.org/10.2307/2987742
[26] Pang-Ning Tan, Michael Steinbach, and Vipin Kumar. 2016. Introduction to data mining. Pearson Education India.
[27] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, Yuanyuan Zhou. 2004. C-Miner: Mining block correlations in storage systems. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'04)*.
[28] Gala Yadgar and Moshe Gabel. 2016. Avoiding the streetlight effect: I/O workload analysis with SSDs in mind. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'16)*.
[29] Miryeong Kwon, Jie Zhang, Gyuyoung Park, Wonil Choi, David Donofrio, John Shalf, Mahmut T. Kandemir, and Myoungsoo Jung. 2017. TraceTracker: Hardware/software co-evaluation for large-scale I/O workload reconstruction. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'17)*. DOI: https://doi.org/10.1109/IISWC.2017.8167759
[30] Search Engine I/O. Retrieved March 2020 from http://traces.cs.umass.edu/index.php/Storage/Storage.

[31] Dushyanth Narayanan, Austin Donnelly, and Antony I. T. Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. In *ACM Transactions on Storage* 4, 3 (2008), 10:1–10:23. DOI: https://doi.org/10.1145/1416944.1416949

[32] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. 2017. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the ACM International Systems and Storage Conference (SYSTOR'17)*. DOI: https://doi.org/10.1145/3078468.3078479

[33] Wenhui Zhang, Qiang Cao, Hong Jiang, and Jie Yao. 2018. PA-SSD: A page-type aware TLC SSD for improved write/read performance and storage efficiency. In *Proceedings of the International Conference on Supercomputing (ICS'18)*. DOI: https://doi.org/10.1145/3205289.3205319

[34] Jun Li, Zhibing Sha, Zhigang Cai, François Trahay, and Jianwei Liao. 2020. Patch-based data management for dual-copy buffers in RAID-enabled SSDs. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3956–3967. DOI: https://doi.org/10.1109/TCAD.2020.3012252

[35] Xiaofei Xu, Zhigang Cai, Jianwei Liao, and Yutaka Ishikawa. 2020. Frequent access pattern-based prefetching inside of solid-state drives. In *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE'20)*. DOI: https://doi.org/10.23919/DATE48585.2020.9116382

[36] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable write cache in flash memory SSD for relational and NoSQL databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'14)*. DOI: https://doi.org/10.1145/2588555.2595632

Received October 2020; revised April 2021; accepted July 2021