

NCache: A Machine-Learning Cache Management Scheme for Computational SSDs

Hui Sun^{ID}, Qiao Cui, Jianzhong Huang^{ID}, *Member, IEEE*, and Xiao Qin^{ID}, *Senior Member, IEEE*

Abstract—Inside a solid-state disk (SSD), cache stores frequently accessed data to shorten the user-I/O response time and reduce the number of read/write operations in flash memory, thereby improving SSD performance and lifetime. Most existing cache schemes anchor in the spatiotemporal locality of I/O requests in workloads. In the face of a long-time workload, high performance and hit rate often get lost in these caching schemes. Flash memory-aware caching schemes trade hit ratio to prolong SSD lifetime. In this article, we advocate for a machine-learning-based caching scheme named *NCache* to optimize both hit ratio and SSD performance. In *NCache*, we construct a machine learning (i.e., ML) model to predict whether data are reaccessed before being evicted from the cache. The cache replacement scheme preferentially evicts data that would not be accessed in the cache. The cache space is conserved for valid data that are likely to be repeatedly accessed. A pipelined scheme is implemented to accelerate the ML model, alleviating the time-cost of *NCache*. A double-linked list boosts the data addressing and cache replacement process. *NCache* is orthogonal to the existing caching schemes within the flash translation layer. The results validate *NCache* under a handful of real-world enterprise traces. Taking *prn_0* as an example, *NCache* reduces the response time of LRU, clean first LRU (CFLRU), GCaR_LRU, GCaR_CFLRU, and LCR by up to 15% with an average of 6.4%. The erase count is slashed by 16% at the maximum. Importantly, *NCache* is adroit at optimizing write amplification by up to 15.9%.

Index Terms—Cache management schemes, machine learning (ML), NAND flash, solid-state disks (SSDs).

I. INTRODUCTION

NAND Flash-based solid-state disks (SSDs) have been widely adopted in large-scale storage systems thanks to SSDs' high performance and low power consumption [1], [2], [3] compared with hard drive disks (HDDs). A portion

of DRAM close to an SSD controller caches user I/O requests from the underlying flash memory. I/O requests sharing the same logical page addresses as the previous ones are serviced by the cache, which shortens the I/O response time [1]. Due to limited cache space, a caching scheme plays a vital role in determining SSD performance and lifetime.

Inside an SSD, improving hit ratio and flash memory endurance becomes a focal point in the development of existing caching schemes. The spatiotemporal locality in workloads is employed to drive up hit ratio. Flash memory-aware caching schemes offer to improve hit ratio and program/erase cycles (P/E) in flash memory according to the status of flash memory, for example, garbage collection (GC) and chip load [4], [5], [6]. However, spatiotemporal locality-based caching schemes are slated to predict hot/cold data through historical I/O access pattern in workloads. These schemes hardly boost performance under weak-locality workloads. The locality of workloads tends to have fluctuation within a time period. If the current workload is featured with strong spatiotemporal locality, a locality-based caching scheme will exhibit a high hit ratio; otherwise, the hit ratio will deteriorate. Thus, it is challenging to guarantee a high cache hit ratio in such a dynamically changing workload. In addition, flash memory-aware caching schemes manage hot and cold data by gleaning historical I/O access pattern. More often than not, discerning hot data from cold data using locality details is inaccurate; thus, the hit ratio is sacrificed to promote the SSD performance.

In this article, we propose a machine learning (ML)-based caching scheme or *NCache*. *NCache* gleans the workload features in real time. Features are taken as input parameters of an ML model. Then, *NCache* judiciously *learns* whether an I/O request is reaccessed from it entering the cache to be evicted out. *NCache* provides a unique and definite decision result for each cache data. *NCache* preferentially replaces data that will be less likely to be accessed. This type of data is referred to as invalid data. Meanwhile, valid data entail that the cache data will be repeatedly accessed (see also Fig. 3 in Section III).

NCache puts forward a handful of salient and distinctive advantages. First, *NCache* leverages workload features in the cache to alleviate the limitations of temporal and spatial localities. The *NCache* aims to proactively learn the I/O behaviors to capture workload features. *NCache* is adept at achieving accurate cache replacement under weak-locality workloads. The locality in I/O requests is employed in the caching scheme powered by a traditional linked list. Thus, *NCache* delivers high accuracy in the invalid/valid data separation. Second, *NCache* is slated to determine future I/O behaviors with regard to each cache data. *NCache* preferentially replaces invalid data

Manuscript received 20 January 2022; revised 29 July 2022; accepted 9 September 2022. Date of publication 22 September 2022; date of current version 19 May 2023. This work was supported in part by the National Natural Science Foundation of China under Grant 61702001, and in part by the State Key Laboratory of Computer Architecture (ICT, CAS) under Grant CARCH201915. The work of Xiao Qin was supported by the U.S. National Science Foundation under Grant IIS-1618669, Grant OAC-1642133, and Grant CCF-0845257. This article was recommended by Associate Editor A. Shrivastava. (Corresponding author: Hui Sun.)

Hui Sun and Qiao Cui are with the School of Science and Technology, Anhui University, Hefei 230601, Anhui, China (e-mail: sunhui@ahu.edu.cn; qiaocui@ahu.edu.cn).

Jianzhong Huang is with the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: hjzh@hust.edu.cn).

Xiao Qin is with the Computer Science and Software Engineering, Samuel Ginn College of Engineering, Auburn University, Auburn, AL 36830 USA (e-mail: xqin@auburn.edu).

This article has supplementary downloadable material available at <https://doi.org/10.1109/TCAD.2022.3208769>, provided by the authors.

Digital Object Identifier 10.1109/TCAD.2022.3208769

that meet requirements of real-time status in flash memory. Hence, NCache is outpacing existing caching schemes in terms of high hit ratio and performance. Notably, NCache is deployed in computational SSDs, which offer computing resources to garner real-time features and to execute ML models. NCache is integrated with the existing caching schemes as a plug-in inside an SSD and, therefore, NCache produces a small footprint coupled with low computing cost.

The contributions of this study are highlighted as follows.

- 1) We put forward a new data organization to record information of cache data (e.g., hit/miss and location at the link list) for the ML model. We unify the software stack for cache management with that for ML to enable directly callback among different libraries, avoiding switch cost between two software stacks. Thus, NCache is orthogonal to existing caching schemes.
- 2) Our proposed ML-based NCache learns workload features in cache systems, thereby optimizing cache replacement. Invalid data are evicted from the cache; meanwhile, valid data are preserved to improve cache performance.
- 3) We forge a pipeline scheme consisting of three processes, namely, feature collection, ML model execution, and I/O-request services. The pipeline significantly shortens NCache's computing overhead.
- 4) We devise a doubly linked list to quickly invoke and organize ML results in NCache. An evicted linked list stores invalid data. Cache data in the evicted linked list is given a high priority to be swapped out. The doubly linked list provides an efficient data address to avert an unnecessary traverse of the entire linked list.

This article is organized as follows. We present the background and related work in Section II. We introduce the motivation in Section III. NCache details are presented in Sections IV and V. We articulate the experimental setup in Section VI. In Section VII, we clarify comparisons between NCache and other caching schemes. Extensive studies are delineated in Section VIII. Finally, we conclude our work.

II. BACKGROUND AND RELATED WORK

A. Background

The average memory access time T_{mem} of a storage system is derived from hit time T_{hit} , miss ratio R_{miss} , and miss penalty P_{miss} [6]. Thus, we have $T_{\text{mem}} = T_{\text{hit}} + R_{\text{miss}} \times P_{\text{miss}}$, where hit time T_{hit} is the time spent in searching cache, and miss ratio R_{miss} is the frequency of cache misses. The miss penalty P_{miss} is the cost imposed by a cache miss. The summation of R_{hit} and R_{miss} is one. In an SSD, cache-miss penalty causes many write/read requests issued to flash memory when cache replacement occurs. Replaced cache data written into flash memory reduces the number of free blocks. A GC [7], [8] is spurred to conduct out-place-update and erase-before-rewrite operations, which wear out the limited P/E's in flash memory, impairing SSD lifetime. Ideally, an efficient caching scheme should be put forth to suppress miss penalty and prolong SSD lifetime.

A large-size cache stores hot data with an intent to cut back data volume written into flash memory [9]. Infinitely increasing cache size is impractical due to the limitation of power

consumption and costs/bit. By and large, caching schemes fall into the following two categories. First, caching schemes employ spatiotemporal locality to enhance hit ratio, including LRU [10], EELRU [11], 2Q [12], LRFU [13], and ARC [14]. LRU gains its popularity in cache management. Data in a linked list is updated by means of inserting data at the head of the list. Newly updated data or hit data moves to the head because the data have a high probability of being reaccessed. Second, caching schemes consider hit ratio and miss penalty in SSDs, such as clean first LRU (CFLRU) [4], GCaR [6], and LCR [5]. CFLRU uses the LRU-linked list and distinguishes clean from dirty data. At the tail of a linked list, CFLRU has a data window, in which clean data is first phased out. This is because the replacement scheme for clean data deletes the cache data rather than destaging data into flash memory.

Caching schemes in the first category aim to evict data based on spatial-temporal locality of I/O requests. Under weak-locality workloads, caching schemes hardly have positive impact on hit ratio. In addition, cache misses cause a penalty to access flash memory; this becomes a challenge to improve both hit ratio and miss penalty. The cache data at the head of an LRU-linked list has higher hit ratio than those at the tail. To reduce the cost of accesses to flash memory, the tail data are evicted with a decreasing hit ratio [4], [5].

B. Related Work

1) *Cache Management Schemes*: Caching schemes can effectively improve the performance of storage systems [9]. Cache data are evicted and updated by replacement schemes [10], [15].

Hit-Ratio-Based Caching Schemes: These caching schemes enhance cache utilization by increasing hit ratio, boosting overall performance. They rely on I/O access patterns, e.g., recency in LRU, frequency in LFU, or the combination of both in SLRU [16] and ARC [14]. In the realm of CPU cache, the Belady algorithm is applied in the past cache access information, which guides future cache replacement [17]. PRP [18] replaces data with the minimum hit ratio rather than the maximum reuse distance.

Flash Memory-Aware Caching Schemes: SSDs have asymmetric read-write performance [19], [20]. Existing caching schemes devised for HDDs are inadequate for SSDs [21], [22]. Caching schemes consider features in flash memory to ramp up SSD performance. CFLRU divides an LRU-linked list into a work area and a first-clean area. The clean data is first replaced while maintaining good hit ratio in flash memory. GCaR profiles chips' GC status in advance. Then, cached data is written into the no-GC chips, avoiding the conflict between GC and user I/Os. Similarly, LCR [5] replaces cache data to chips with light load. ECR [23] optimizes LCR to alleviate harmful effects of adjusting a write window. Fab [24] organizes pages that belong to the same erased block. These pages are managed in an LRU-linked list. When data in a group hits cache or just arrives, the entire group moves to the linked list's head. Dirty data is written in flash memory whereas the clean data is discarded.

Unlike the aforementioned caching schemes, our NCache employs the ML model to govern data replacement. NCache

TABLE I
LOCALITY OF TRACES

Trace	Total I/Os	Read Size (GB)	Write Size (GB)	Avg.size (KB)	Avg.LPA ¹	Standard Deviation ²	Temporal Locality ³	Space Locality ⁴
hm_1	609,311	8.26	0.54	7.99	84,769	117,377	25%	93.8%
WebSearch1	105,5448	15.24	0.0017	15.1455	2,826,307	1,220,527	0	21.8%
proj_0	4,224,524	8.97	144.27	38.03	1,666,259	1,434,655	6.4%	33.8%
prxy_0	12,518,968	3.04	53.79	4.761	510,627	966,618	5.6%	42.8%
stg_0	2,030,915	7.33	15.09	11.57	815,675	623,726	10.4%	68%
prn_0	5,585,886	13.12	45.96	11.09	7,159,110	7,091,273	6%	93.8%
financial1	5,334,987	2.652	14.57	3.384	51,537	121,220	9.6%	82.8%

¹Col.6 is the average of logical page addresses in traces. Symbol 'LPA' represents logical page address. ²Col.7 is the standard deviation of logical page addresses in traces. ³Col.8 is the temporal locality of the trace logical addresses, and it reflects the probability that requests will be repeatedly accessed within a period. ⁴Col.9 is the space locality of the trace logical addresses, and it represents the probability that an already accessed request will be accessed again for its adjacent logical addresses.

drives up performance by servicing valid data at low cost. NCACHE is easy to integrate with other caching schemes.

2) *Machine Learning in Storage Management*: Applying ML models in storage systems grasps huge popularity. An accelerator was integrated in an SSD [25] to deploy deep learning models to complete data retrieval and analysis in flash memory. Designers provided a method to tradeoff lifetime, performance, and energy consumption in the nonvolatile memory medium. MCT [26] was proposed to employ an ML model to dynamically balance these metrics. An ML model, predicting CPUs' reuse data [27], asserts that most modern memory prefetchers rely on spatiotemporal locality to speculate future memory addresses. Irregular data structures are widely used in emerging workloads, which show a low degree of spatiotemporal locality. Thus, locality-aware caching schemes may become inadequate for existing memory prefetchers. A context-based reinforcement learning memory prefetcher was built to guide memory access semantics according to the features of machine and application codes. The support vector machine (SVM) [28] was applied in a hardware predictor structure for cache replacement. A random forest tree-based caching [29] is designed to achieve optimal cache decision by prediction the popular Web content for proactive caching.

To enhance cache hit ratio and extend SSD lifetime, Wang *et al.* [30] introduced an ML model to avoid unnecessary writes in server cache. In contrast to the above studies, we utilize an ML model to predict whether or not future I/O accesses hit cache. Our solution is adroit at filtering data that enters (or evicts) the cache. Traditional caching schemes are guided by the Belady algorithm [31] to improve cache performance; however, it is a challenge to capture future situations. Using spatiotemporal locality to simulate future situations is ineffective. In SSDs, caching schemes should consider flash memory characteristics while trading hit ratio for flash memory endurance and SSD lifetime. As such, caching schemes face a predicament in optimizing SSD performance. ML makes it possible to predict future accesses without relying on historical information.

To address the issues in the spatiotemporal-locality-based schemes, we develop an intelligent caching scheme, NCACHE, to leverage ML algorithms in computational SSDs. NCACHE is slated to be incorporated into most current caching schemes with little cost.

III. MOTIVATION

We place seven realistic enterprise traces (see Table I) under the microscope to investigate the spatiotemporal locality of I/O requests. We analyze the spatiotemporal locality at intervals of 500 consecutive I/O requests in each trace, aiming to unfold correlations among I/O requests. *Temporal Locality*: A trace is divided into multiple areas, each of which is comprised of 500 consecutive I/O requests. We count the number of same logical addresses between two neighboring areas, the metric of which is counted. The average ratio of the total of these metrics to entire logical addresses is defined as *temporal locality* of the trace. *Spatial Locality*: 500 I/O requests in an area are divided into subareas by 128 pages in flash memory. We assume that 500 I/O-requests logical addresses are partitioned into N subareas. In the following 500 I/O requests, we record the number of I/O requests that belong to these N subareas. Accordingly, we register the number of I/O requests in the same subarea between two consecutive areas. The average ratio of these I/O requests to entire ones is computed as *spatial locality*.

In Table I, the *standard deviation* of LPAs captures the fluctuation of I/O requests' logical addresses. The larger the value of time locality is, the easier I/O requests will be accessed. The temporal locality in hm_1 and WebSearch1 is 25% and 0, respectively. Thus, hm_1 is easier to be accessed. The spatial locality of hm_1 and WebSearch1 is 93.8% and 21.8%, which assert that the probability of neighboring I/O requests to be accessed is 93.8% and 21.8%. WebSearch1 has a poor spatiotemporal locality. Our empirical study suggests that performance improvement offered by the traditional caching schemes has hit a roadblock under WebSearch1.

We randomly select subarea-range I/O requests to dabble in the spatiotemporal locality of traces. Fig. 1 unveils that prn_0 has a sharp fluctuation, followed by the WebSearch1 and proj_0. The rest of traces have smooth fluctuation. The *avg.LPA* and standard deviation in prn_0 are the biggest among its peers; thus, prn_0 has a sharp fluctuation. Although WebSearch1 shows a slight oscillation, most LPAs are not at the same y-axis value. This trend indicates that I/O requests are relatively discrete. Thus, I/O requests that have been serviced in the prior stage are less likely to be issued again. On the contrary, the LPAs in prn_0 are close to the same y-axis value, which means that the I/O requests tends to be repetitive. Prn_0 has a better spatiotemporal locality than WebSearch1. Compared with WebSearch1, prn_0's I/O requests fluctuate but

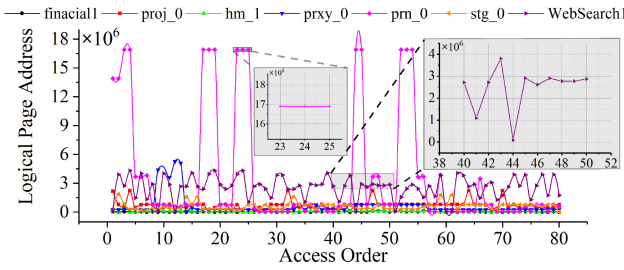


Fig. 1. Spatiotemporal locality of I/O requests in workloads.

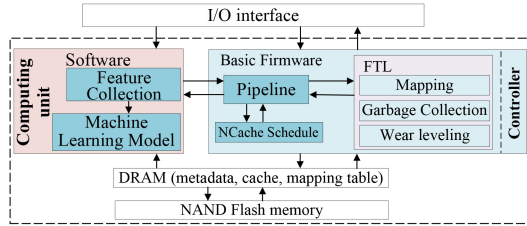


Fig. 2. Overview of the NCACHE system.

amount of I/O requests in the same LPA range are concentrated in the peak and trough stages.

In the face of weak spatiotemporal-locality workloads, the predicted result of caching schemes become inaccurate. The decision result can judge the access probability for I/Os. Caching schemes relying on the least recently used algorithm reconcile the cost of I/Os accesses to the flash memory. A raft of prior work [32], [33] predict reused data in CPUs, where the performance is optimized by ML modules [27], [34]. The re-access distance is employed to obtain a high hit ratio [31]. The mechanism of bypass caching schemes are studied to improve cache performance [27], [33], but no standard decision mechanism is forged to determine which data is written to the cache. A one-time-access standard method [30] is designed to filter unnecessary data in the cache to life cache efficiency.

We are motivated to propose an ML-model-based caching scheme entitled *NCACHE* to alleviate optimization issues in the spatiotemporal locality-based caching schemes. The ML model is slated to predict whether cache data are likely to be accessed in the future. A data item enters the cache regardless of its future accesses. We leverage a management method to configure data residence time in the cache, where data are divided into valid data and invalid data. Valid cache data will be accessed in the future whereas invalid cache data that have no luck to be accessed again in the future. Our cache replacement scheme aggressively evicts invalid data from the cache, thereby boosting cache hit ratio and preventing deterioration in flash memory.

IV. SYSTEM OVERVIEW

Fig. 2 depicts the ML-based NCACHE implemented inside a computational SSD, which is furnished with a computing unit to perform the ML model and feature collection inside the SSD. The I/O interface module receives user I/O requests converted into a page-size data. By using page request information, the feature collection module harvests workload features—the input parameters of the ML module. The ML module analyzes the features and makes decisions to assist

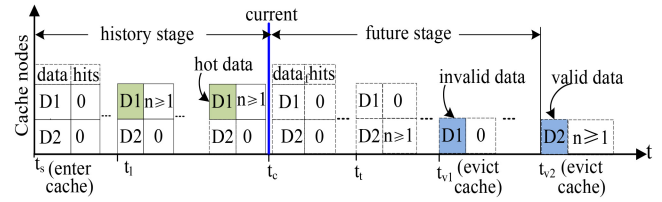


Fig. 3. Discrepancy among hot data, invalid data, and valid data.

NCACHE in dividing cache pages into valid data and invalid data. The cache replacement module preferentially destage invalid data into flash memory. NCACHE stores valid data in the cache as much as possible. The flash translation layer (FTL) manages flash memory, including address mapping, GC, and wear-leveling. The pipeline module performs feature collection, ML module, NCACHE scheduling, and FTL in parallel. If only cache service is sparked (i.e., cache hit), the ML module and the caching scheme are concurrently performed in parallel. When the cache is full and cache misses, the ML model, caching scheme, and FTL can perform in a pipeline manner.

In Fig. 3, at time $t = t_s$, user data D1 and D2 enter the cache. At time $t = t_l$, this is the current state of the cache that data D1 has been accessed more than once; then, there are n ($n \geq 1$) cache hits for D1. At time $t = t_c$, D1 is a hot data according to its historical cache hits. The term “ f hits” denotes the number of cache hits for cache data from time t_c to the point when the data are evicted. At time $t = t_c$, the future hit ratio (f hits) of data D1 and D2 is set to a default value of zero. At time $t = t_{v1}$ and t_{v2} , these are the current states of the cache replacement occurrence for D1 and D2. If there are n cache hits for D2 before it is evicted, data D2 is marked as valid data (i.e., f hits ≥ 1), whereas D1 is invalid because D1 has not been accessed (i.e., f hits = 0).

We employ valid data and invalid data as the reference metrics for classification prediction results in NCACHE: metrics valid and invalid correspond to “label = 1” and “label = 0” as articulated in Section V-B. NCACHE predicts I/O accesses in the cache in advance based on workload features. Then, we determine the data in the future I/O access patterns to be valid or invalid. Amid cache data replacement, NCACHE strives to evict invalid data (e.g., D1) in favor of accommodating valid data.

In addition, we shed bright light on the discrepancy between hot data and valid data. Hot data are discovered from statistics of I/O access patterns in workloads. Hot data are elected from past-access features, whereas valid data are detected from future-access predictions. For example, hot data D1 is frequently accessed during a time interval from it entering cache (time t_s) to the current time ($t = t_c$). In the LRU, hot data are determined by an interval (the number of I/O requests) between two consecutive accesses. The smaller the access interval is, the hotter the data item becomes. Unfortunately, this method is ineffective when I/O access patterns dynamically change; the traditional scheme fails in swiftly identifying such dramatic pattern changes. When it comes to hot data, frequent I/O accesses in the past entail a high priority. However, if hot data are not accessed for a long time, the data are evicted in the future, but the data occupy the limited cache space for a long period. From time t_c to time t_{v2} , D1 has not been accessed

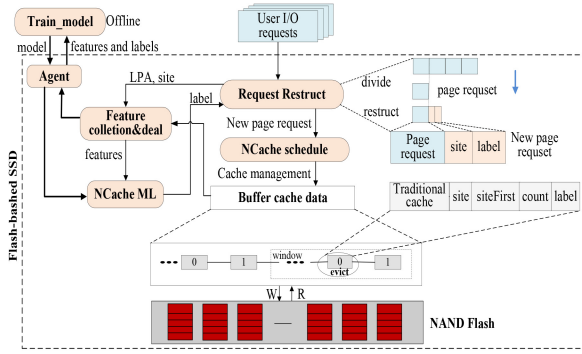


Fig. 4. NCache design of processing I/O requests.

before it is evicted; thus, data D1 is envisioned as invalid data. Our proposed valid and invalid data that are related to future I/O access patterns in the workload rather than the hot data.

V. DESIGN OF NCACHE

Fig. 4 plots a procedure of serving an I/O request in NCache. The request restructure module is an extension of the host interface logic module (HIL).¹ As the first step, this module divides an I/O request into page requests according to the page size in flash memory. A new page request is composed of two fields, namely, label and site. The feature collection & deal module obtains the LPA and serial number of a page request. Then, this module gleans the page-request features based on *frequency*, *site*, *recency*, *timetolive*, and *lifetime*, which are transmitted to the ML module. The ML module in turn generates label information for the page request. The label information and serial number is used to update the label value and site value of the new page request. The NCache scheduler undertakes cache replacement.

Notably, the ML model training is an offline procedure, which imposes no effect on SSD performance. The ML model is executed when the SSD runs workloads. Updating an ML model while SSD is handling requests is abominable because updating the ML model and data write operations consume internal bandwidth that impairs user performance. For this reason, we opt for updating the ML model when SSDs are sitting idle.

A. ML Model in NCache

The decision-tree algorithm [35], [36], a supervised ML algorithm, offers high precision and small footprint cost. Thus, we adopt a decision-tree model in NCache.

We choose an ML method with high precision and low complexity to predict I/O behaviors of the cache. The ML models, including SVMs, decision tree, and random forest, are popularly applied to uncover I/O patterns in workloads. Regarding the choice of ML models, we ought to make tradeoff among model size, accuracy, computational cost, and interpretability because the ML model is deployed in SSDs furnished with limited resources. A decision tree is featured with lightweight, adequate accuracy, and less computational cost than the SVM and random forest counterparts. On the interpretable side, the

decision tree catered for classification is easily understood either in the graphical form or a set of “If-Then” rules converted [37]. The decision tree can summarize the classification rules from the samples of a training set, and these rules are used to classify data, which in turn predict future I/O access patterns. In a prior study [30], the performance of several ML algorithms is evaluated under realistic traces: the results reveal that the decision-tree algorithm is the winner.

In a decision tree, each nonleaf node represents a test on a feature attribute; each branch represents the feature attribute’s output in a certain value range; each leaf node stores a category. From the root node, the decision-tree-based decision-making process tests feature attributes in the items to be classified, the result of which is used to ensure the output branch until it reaches the leaf nodes. The category in the leaf node is recorded as a decision result.

According to probabilities of various cases, the decision-tree algorithm projects feature probabilities in each sample data. The decision tree, aiming to analyze the semantics in a sample data, extracts a series of rules from a large-scale data to figure out features in the sample data set. The decision-tree algorithms include *ID3* [38], *C4.5* [39], and *CART* [35]. The *CART* tree employs Gini index to evaluate the quality of classification, which simplifies the computing process. A decision tree generally uses information gain or information gain ratio for classification, but the two metrics have logarithmic operations bearing extra overhead. The *CART* tree applies the Gini index to facilitate feature classification. The Gini index performs in a similar fashion as that of the information gain and information gain ratio without logarithmic operations, yielding fast training results. By analyzing historical I/O access patterns in workloads, we obtain a classification model in the ML framework to unfold I/O access patterns hidden in workloads. According to the features of I/O requests, the ML model provides classification results predicting I/O access patterns.

As the first step, we should determine features and labels: we extract five features from a page request before it enters the cache, and a request label is gleaned after data is evicted from the cache. We employ the decision-tree algorithm to learn the relationship (model) between features and labels during the training phase, in which the rules are forged for classification. Then, we obtain the ML model in NCache. Next, we predict the future I/O access patterns in workloads: in the inference, we take the features of a page request as input parameters in the ML model, which foresees the values of the label in the page (label = 0 or label = 1). When the label is zero, the cache page request is forecasted as invalid data that will not to be accessed before it is evicted; then, the data can preferentially be evicted during the cache replacement. If the label is 1, the cache page request will be valid data that will be accessed more than once in the future before being evicted. We use the cross-validation method to generate training sets and test sets from several realistic traces. In this work, we split the datasets up into 20% training and 80% testing sets. To prevent overfitting, we limit the depth of the tree.

B. Feature Extraction

Feature: The feature collection & deal module extracts workload features. On the basis of LPA and site of a page request, this module harvests features to be transferred to the

¹In an SSD, the device-side system includes the HIL, FTL, and NAND flashes. When arriving at the SSD, the I/O request is inserted into the request queue in the HIL that divides the request into page-sized subrequests.

ML module. These features are also sent to the agent module as a part of the training dataset. The range of feature is limited to data's lifetime in the cache. During the lifetime of cache data, the feature collection & deal module collects the following five features.

Frequency is the number of continue accesses to the same LPA of the page request in a period. This term—reflecting the hot/cold feature of I/Os—assists cache management for incoming data.

The *size* of an user-I/O request. Large user-I/O requests are divided into page-sized requests inside an SSD. The order of LPA in these pages sequentially increases; the sequence length affects the cache hit ratio. Innumerable partitioned page requests that have a long consecutive-sequence length can take up large cache space. When the LPA of a new request is largely different from those of data residing in the cache, the workload has a weak spatiotemporal locality giving rise to a declining hit ratio. Otherwise, the workload exhibits a good spatiotemporal locality, which strengthens the cache hit ratio. The sequence length and the spatiotemporal locality in workloads directly affect the hit ratio.

Recency is the number of I/O requests between the last two consecutive access hits (read/write) to a cache data. A large recency value signifies that the cache data have not been recently used.

Timetolive is the number of I/O requests from the data in the cache to the current accessed I/O request.

Lifetime indicates the minimum number of an LPA that is evicted from the cache. *Timetolive* and *lifetime* denote the serial number of the LPA in the lifetime. With a large timetolive and a small lifetime, the possibility of cache data being accessed is small; thus, the cache data is trending toward invalid data.

Notably, data statistics and calculations for duration time and time intervals of requests will be frequently assessed. Using “time” to calculate the two metrics will bring high overhead and inconvenience. For example, the timetolive depends on the interval between the current time and the time that a request enters the cache. For the lifetime, we need to estimate the time interval of cache data from current time to the point when data are evicted. But it is difficult to predict eviction time owing to frequent data replacement in cache. Hence, we reckon that it is challenging to make use of time to gauge the statistics for timetolive and lifetime. We advocate for employing the number of requests rather than time to obtain the two features in NCACHE.

Feature Extraction Method: An I/O request received by the HIL module is referred to as an *original* request. If the request's size exceeds one cache page, the request restructure module in HIL divides the request into multiple page-sized requests. We propose in NCACHE a new way of organizing I/O requests. Each page request has two data fields—1) *site* and 2) *label*. The *site* field denotes the serial number of the corresponding page request; the term *label* represents whether page request LPA will be accessed in the future. Specifically, “1” means that the page will be accessed, whereas “0” indicates that the page will not be retrieved. In addition, these multiple partitioned page-size requests are sequential; thus, we introduce a page-request counter to register the number

TABLE II
NOTATIONS AND DESCRIPTION

Notation	Description
S_{cr}	the serial number of the current I/O request.
S_{cbi}	the serial number of LPA in the trace corresponding to the i^{th} node in the linked list that is latest cached.
S_{obi}	the serial number of LPA in the trace corresponding to the i^{th} node in the linked list that is first accessed.
C_i	the number of hits of the cache node i , the initial value is 1.
R_MAX	default maximum value of recency.
i	the position of the node at the linked list from the tail to the head.

of page requests. the request restructure module assigns the count values to the data field site in a newly reorganized page request. The page request provides *LPA* and *site* to the feature collection module. This results of the ML model are assigned to the data field *label*.

Cache Data Organization: The feature collection procedure consumes time; thus, we bring forth a novel mechanism to capture the workload features to cut back footprint and run-time cost. Data have a lifetime in cache before being evicted. To record dynamic features during the lifetime, we devise a lightweighted data organization to store feature changes in the cache because the features alter with cache replacements. With the data organization, NCACHE collects features in real-time without impairing system performance. We extend the data organization by four fields—1) *site*; 2) *sitefirst*; 3) *count*; and 4) *label*. *Site* denotes the serial number of the latest page request that accesses to the cache data. *Sitefirst* records the serial number of the first access request to the cache page in its lifetime. *Count* is the total number of LPA occurrences in the cache. *Label* is a decision regulated by the ML model.

Feature Processing: To demonstrate feature extraction, we analyze five I/O-request features, including *frequency*, *size*, *recency*, *timetolive*, and *lifetime*.

The *frequency* is denoted as ($\text{frequency} = C_i$ or 0). Table II lists notation and description. When an LPA hits cache, the frequency value is equal to C_i ; otherwise, the value is 0. The data field count—a counter in the cache—records the number of occurrences of LPA for the current cache data. Once data enters the cache, the count is increased by one when it hits cache. When request L_i hits cache, count C_i is the number of cache hits for request L_i in the past.

For *recency*, we have ($\text{recency} = S_{cr} - S_{cbi}$ or R_MAX). When the LPA hits cache, recency is the discrepancy between S_{cr} and S_{cbi} ; otherwise, the value is R_MAX . The cache site records the serial number of the last cache hit. When a new request hits cache again, its site is a new value compared with that of cache data. Meanwhile, the distance between the last two consecutive accesses is the difference between request site S_{cr} and site S_{cbi} in the i th cache data.

There is $\text{timetolive} = S_{cr} - S_{obi}$ or 0. When the LPA hits cache, the *timetolive* measure is the difference between S_{cr} and S_{obi} ; otherwise, the value is zero. The *sitefirst* value (i.e., S_{obi}) in the i th cache data records its serial number in the cache when it is first placed. Current request L_i hits cache again. The *timetolive* corresponding to the cache data is the

difference between the *site* of request L_i (S_{cr}) and the *sitefirst* of i^{th} cache data (S_{obi}).

The *lifetime* is expressed as ($\text{lifetime} = i$ or 0). When the LPA hits cache, *lifetime* is gauged as i ; otherwise, the value becomes zero. This term indicates the distance between the cache data hit by request L_i and the end in the linked list. The value can be acquired by traversing the linked list.

The term *size* is the size of an I/O request in the workload.

Training Datasets and the ML Model: In the ML model, we obtain labels as output when provided with features of I/O requests as input. The feature collection & deal module gleanes the features and LPA from each I/O request. The cache hits of a page request are defined as labels corresponding to these features. The feature and label are asynchronous because it is opaque whether or not an I/O request hits cache after feature collection. The messages of cache hit/miss become available after data are evicted from the cache. Recall that a cache data contains features *sitefirst* and *site*. When data are evicted from the cache, two features are acquired. When *sitefirst* and *site* values disagree, it asserts that the replaced data hits cache at least once. In this case, two pieces of information is gathered: 1) the label of page request *sitefirst* is 1 and 2) the label corresponding to the page request *site* is 0. If the *sitefirst* and *site* values are identical, the label corresponding to the *site* is 0.

Notably, the terms *site* and *sitefirst* are the serial numbers of request pages and the serial numbers of features. The order of features of I/O requests are consistent with those in the workload. Thus, the I/Os-request location information of labels (*sitefirst* and *site*) is the same as location of the corresponding features in the workload. The *train_model* module delineates a range in the feature information through the *sitefirst* and *site*. There is a consistent one-to-one match between features and labels; thus, features and labels of requests are assembled as a dataset. We train the ML model using the dataset in the offline mode. The model is incorporated to the NOR Flash through an agent module. The NCache scheduler handles requests using this model.

C. ML Model Inference

The feature collection & deal module discovers workload features, which are handed over to the ML module for data analysis. NCache employs CART trees for data classification. We set the tree's maximum depth to five, which avert overfitting in the ML model.

There are two approaches to invoking an ML model. First, a decision-tree model is converted into native code with zero dependencies in a C runtime environment because the model is a dictionary converted into an "if-else" structure to make the decision. Second, Python functions are invoked in the C runtime environment; afterward, the model is performed. The first approach directly calls parameter values in the native code, which satisfies the real-time requirements in applications. The second one, which depends on the ML frameworks and dependency libraries, passes parameters between the Python and C runtime environments, expanding execution overhead. In addition, it is expensive to deploy the Python library in computational SSDs with limited resources. Thus, the former method is elected in NCache.

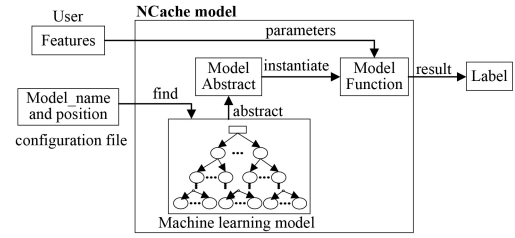


Fig. 5. ML model in NCache.

Fig. 5 shows the procedure of calling the ML model from the firmware. The model is stored in NOR Flash memory in the form of a dictionary. It can be updated through the agent module in Fig. 4. When initializing the SSD or updating the ML model, the firmware can find the ML model according to its name and location in the configuration file. The model abstract module converts the ML model into a tree structure that is instantiated as a model function in the firmware. It must be noted that the ML model is abstracted and instantiated in the stage of SSD initialization and model updating. Classification conditions of the decision tree are converted into the "If-Else" conditions in the model function. Leaf nodes are referred to as the return value of the function. The feature of requests is set as parameters of the function. Then, the function outputs the judgment result.

Because the model is converted into a function, the computational cost of the model is negligible. The extra cost of NCache mainly includes the feature collection module. We use the cache to obtain feature of requests through a customized data structure; thus, the feature collection of requests is transformed into an access to the cache. We design a pipelined mechanism to reduce the feature-collection cost (see Section V-E).

The model outputs two values (0 and 1). 0 indicates that the LPA of a page will either be accessed after a long time or not be accessed at all. 1 signifies that the LPA will soon be accessed again. Each page is given a unique result through the ML inference. NCache accurately evicts cache items marked with 0 to free up space for data labeled with 1. To improve the hit ratio, NCache evicts cache items with 0 in line with the flash state (e.g., GC). Although the ML model discerns valid data from invalid ones, it is prudent to assign priority levels to the valid (or invalid) data. An LRU-linked list reflects the priorities of cached items. In a linked list, these cache items with 0 and 1 may be staggered but the priority of the resident cache item with 1 must be higher than that of 0. The priority of a cache item close to the list's head is higher than that of an item near the list's tail. To enhance the efficiency, we develop a doubly linked list in NCache to manage cache data.

D. Cache Replacement

When the request restructure module assigns a value to the label, NCache serves I/O requests. Fig. 6 depicts that the cache replacement is orchestrated by the new cache data organization accompanied with the new linked list structure.

The Cached Linked List Structure: The ML model predicts and registers the predicted result for each cached item, and

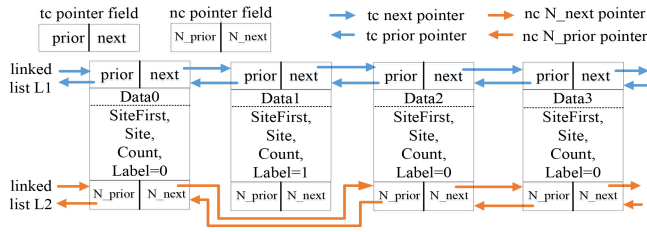


Fig. 6. Doubly linked list in NCache.

the cache linked list uncovers the priority of data access: the tail node has the lowest priority. NCache provides priority to searching invalid data, which may not be a tail node in the traditional LRU. Then, we ought to traverse the linked list to access invalid data. It is time consuming to traverse a linked list, but this problem is solved by implementing a doubly linked list, including linked list L1 and linked list L2 in NCache. Let *LinkL1* and *LinkL2* represent linked list L1 and linked list L2, respectively. We deploy *LinkL2* to manage invalid data in an LRU manner.

Fig. 6 plots the data structure of NCache's linked list, where a cache data item has two data fields: 1) *tc* and 2) *nc*. Field *nc* stores pointers in traditional cache data (pointers next and prior), whereas field *nc* holds newly added pointers for NCache (pointers *N_next* and *N_prior*). The four pointers determine a node position. The pointers next and prior are referring to the next- and prior-location cache data, respectively. These nodes comprise the *LinkL1*, which contains all data nodes in the cache. In contrast, pointers *N_prior* and *N_next* are used for invalid data nodes that will not be accessed according to predictions made by NCache. These cache data nodes form *LinkL2*. When cache data are replaced, the nodes in *LinkL2* are preferentially evicted.

Discussion: We demonstrate the overhead of the doubly linked list and pointer manipulations in NCache. First, the doubly linked list induces space overhead. Each cache item has four numerical values, three integers, and one bit. It is supposed that the cache size is 1 MB while the page size is 8 kB. Each integer occupies 4 B; then, cache data consumes an extra 12 and 1-bit cost. The space consumption caused by the new data organization is 1552 B. Second, we directly locate invalid data rather than traversing the linked list. The access performance of NCache is equivalent to that of LRU. In addition, the pointer manipulations are merely to assign extra *N_prior* and *N_next* pointers to nodes in the linked list. The overhead of the pointer assignment manipulations is minimal, which can be ignored in NCache.

The Window Mechanism: There are two access patterns for cache data in NCache—1) accessed and 2) nonaccessed. Although a data item has recently been loaded into the cache, it may be evicted when its corresponding label marked as 0. The inaccurate results of NCache can impair the cache performance; thus, we put forth a window mechanism. NCache cache replacement is only performed within the window. When there are no cache data in the window that meet the conditions for replacement, the traditional replacement scheme is launched. The node from *LinkL2* is first selected. If *LinkL2* is not empty, a data node at the linked list's tail is chosen. Otherwise, there is no on-demand data in the window; then,

Algorithm 1 NCache-Based Cache Replacement Algorithm

Require: the user page requests *LPA1*, *LPA2*, ... *LPAi*...

```

1: procedure NCACHE-BASED SCHEMES
2:   if LPA hit node i in data cache then
3:     Update_linked_list(LinkL1)
4:     if page request is write request then
5:       Update_data_node(i)
6:     end if
7:     if Node i is within the window then
8:       Update_linked_list(LinkL2)
9:     end if
10:  else
11:    if Buffer is full then
12:      if LinkL2 is empty then
13:        Evict_tail_data(LinkL1)
14:      else
15:        Evict_tail_data(LinkL2)
16:      end if
17:      Write_to_buffer(LPAi)
18:      Update_linked_list(LinkL1, LinkL2)
19:    else
20:      Write_to_buffer(LPAi)
21:      Update_linked_list(LinkL1)
22:      if Affects LinkL2 management data range then
23:        Update_linked_list(LinkL2)
24:      end if
25:    end if
26:  end if
27: end procedure

```

we select a new data node from *LinkL1*. We delineate the caching scheme in NCache as follows.

Algorithm 1 shows the process of NCache schemes for page requests. For a page request *LPAi*, NCache first checks whether it hits cache. If it hits a cache node *i*, NCache updates *LinkL1*. The hit node *i* moves the head of the linked list. If the request is a write, the node *i* needs to be updated. *LinkL2* manages invalid data in the preset window. When node *i* is also located in the window, the *LinkL2* is updated. If the cache misses, when the cache has free space, the data is written into the cache, and *LinkL1* is updated. If the data is invalid and the number of cache nodes is smaller than the window size, *LinkL2* is be updated; otherwise, *LinkL2* does not need to be updated. If the cache is full and *LinkL2* is no empty, the tail node at *LinkL2* is evicted. If *LinkL2* is empty, the tail node at *LinkL1* is evicted. Meanwhile, the new data is written into the cache, and *LinkL1* and *LinkL2* are updated.

E. Pipelined NCache

NCache introduces the overhead cost of the feature collection and ML model computation processes (FC_ML for short) in cache management. The FC_ML is given a high priority at the beginning of each page request service. The cache-hit procedure of NCache embraces FC, ML, judging cache hit/miss, data update, and linked link update. In the case of cache misses, we have FC, ML, judging cache hit/miss, evict, write back, data update, and linked list update. Herein, the write requests are taken as an example within a full cache. A request may be divided into multiple page requests. In the nonpipelined NCache, page requests are sequentially served, causing high time overhead as illustrated in the upper part of Fig. 7. We advocate for a computing pipeline to accelerate NCache's speed. When NCache handles the first-page request in an I/O request, the priority is given to the FC_ML process.

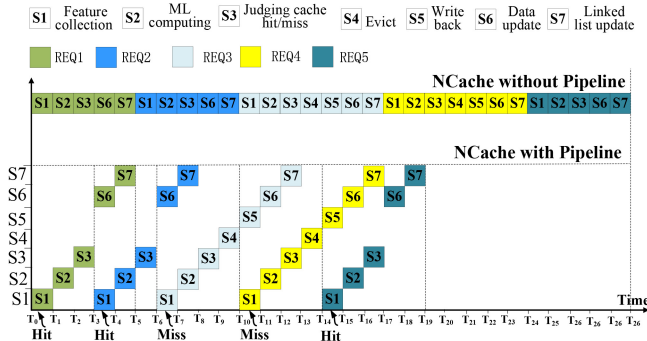


Fig. 7. Processing of requests in the pipelined and no-pipelined NCache schemes. REQ i represents the i th request. S1–S7 denote pipelined stages.

The pipeline scheme is adopted for the feature collection of the subsequent pages. According to the hit-miss status of the current page and its following one, we have four cases—1) (hit, hit); 2) (hit, miss); 3) (miss, hit); and 4) (miss, miss). In Fig. 7, REQ1, REQ2, and REQ5 are cache-hit cases, whereas REQ3 and REQ4 are cache-miss cases.

(Hit, hit): Upon the completion of FC_ML for REQ1, FC_ML of REQ2 is running in parallel with the cache service (stages S6 ~ S7) for REQ1. While collecting data field lifetime in feature collection, the linked list is traversed. During REQ2's collecting data field lifetime, the system figures out whether REQ1 impacts the feature collection process of REQ2. If REQ1 has a smaller lifetime than REQ2, the process of REQ1 affects that of REQ2. The approach to eliminating influence is to minus 1 for lifetime value.

(Hit, miss): When FC_ML of REQ2 is completed, REQ3 starts to perform FC_ML that is in parallel with REQ2's cache service (S6 ~ S7). Because REQ3 misses, the feature collection is initialized without special processing.

(Miss, miss): When FC_ML of REQ3 completes while the cache replacement item is selected, FC_ML of REQ4 are concurrently carried out with the cache services of REQ3 (S5 and S6), which conducts a cache replacement. Deciding whether REQ4 hits or misses is undertaken after the cache replacement. REQ4 misses in this case and; then, the feature collection is kicked off.

(Miss, hit): The FC_ML of REQ4 finishes; meanwhile, the cache replacement item is fixed. The FC_ML of REQ5 starts in conjunction with REQ4's cache services (S5 and S6). When the cache replacement affects the data field lifetime in REQ5, the impact from REQ4 needs to be mitigated by decreasing the lifetime value by one.

Fig. 7 illustrates that the pipelined NCache has a performance edge over the nonpipelined NCache.

VI. EXPERIMENTAL SETUP

Testbed: We employ a popular event-driven SSD simulator—SSDsim—to validate NCache (see Table III). SSDsim accurately simulates the components of an SSD. We have ten configured cache sizes in experiments. NCache is focused on write-intensive workloads. Thus, we selected five traces with high-ratio write requests (proj_0, prxy_0, prn_0, fin1, and stg_0) and two high-read-ratio traces (hm_1 and Websearch1) in Table I to study the performance of NCache.

TABLE III
CONFIGURATION PARAMETERS OF SSDSIM

SSD Organization	Capacity: 16 GB, two channels, two chips per channel
Flash Micro-architecture	8-KB page, 492-Bytes metadata per page, 128 pages per block, 1024 blocks per plane, two planes per die, two dies per chip
Flash latency	Read latency: 25 μ s, Program latency: 200 μ s, Erase latency: 1.5ms.
FTL	GC Threshold: 0.3, Mapping scheme: page-level mapping.

Alternative Caching Schemes: We implement NCache as well as five existing caching schemes. LRU is a widely popular cache replacement scheme. The CFLRU is designed for flash memory, replacing clean data to minimize replacement cost. In this test, we set the write-back window to $C/2$, where C is cache capacity. CFLRU leverages the read-write cache mode and write-back methods. The GC awareness replacement policy (GCaR) is one of the latest caching schemes. Load-aware cache replacement (LCR) considers a chip load in cache replacement.

Performance Metrics: We apply five metrics to quantify the performance of NCache. *Average response time (Avg. RT)* measures an SSD's performance from the perspective of individual I/O requests. This metric is the time interval between an I/O request entering the SQ queue and the host receiving feedback from the SSD. *Write amplification (WA)* is a ratio between the amount of data written to the flash memory and the amount of data issued by a host. A low value indicates that the amount of data written to the flash memory is small. WA reflects an impact of the tested cache schemes on SSD lifetime. *Erase count* is the most conspicuous manifestation of SSD lifetime. Each block in flash memory has a maximum number of erase operations. *Write/Read hit ratio (Write HR/Read HR)* is the probability of an I/O request hitting the cache. *Valid data ratio* is the proportion of cache data that will be accessed in the future when cache replacement occurs.

VII. PERFORMANCE EVALUATION

We conduct experiments to validate the performance of NCache in terms of avg. RT, erase, WA, and hit ratio.

Average Response Time (Avg. RT): Fig. 8 unveils that NCache has an edge over the existing caching schemes. Under proj_0, prn_0, financial1, prxy_0, and stg_0, the NCache-based caching schemes reduce avg. RT by up to 36.0%, 15.4%, 40.0%, 64.9%, and 16.2%. Under hm_1 and Websearch_1, the NCache reduces the avg. RT by up to 56.7% and 12% in the 8- and 512-MB cache.

The avg. RT drops as the cache size increases. When it comes to high spatiotemporal-locality traces (hm_1), the avg. RT changes with the varying cache size. When the cache size is larger than 8 MB, the avg. RT decreases slowly with the increasing cache. For proj_0 and stg_0 with small spatiotemporal locality, the avg. RT hardly changes with a small-sized cache, but it significantly decreases with 128-MB cache. For WebSearch_1 with weak locality, the avg. RT has a little increment with the large cache size. For hm_1, the 2-MB

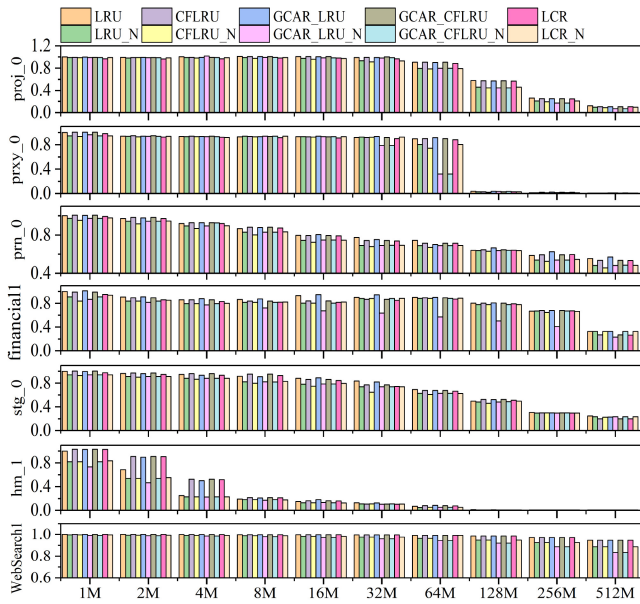


Fig. 8. y-axis represents Avg. RT, normalized to that of the LRU scheme under each trace. The x-axis denotes cache sizes.

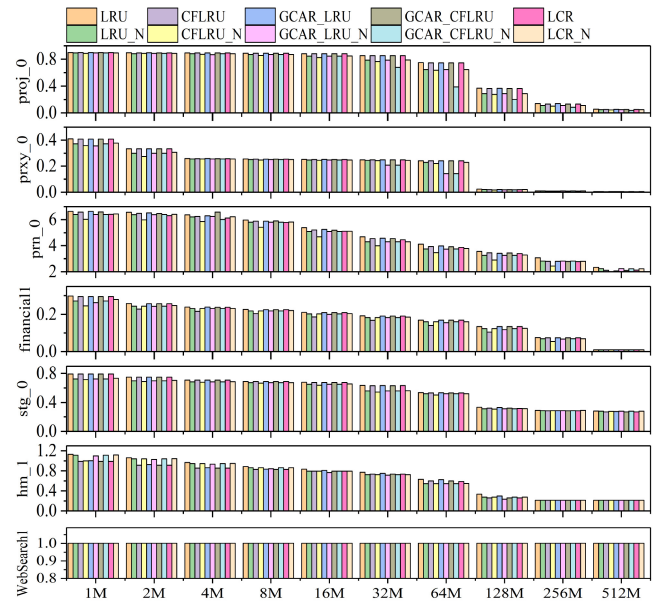


Fig. 10. y-axis denotes WA under different traces and cache sizes.

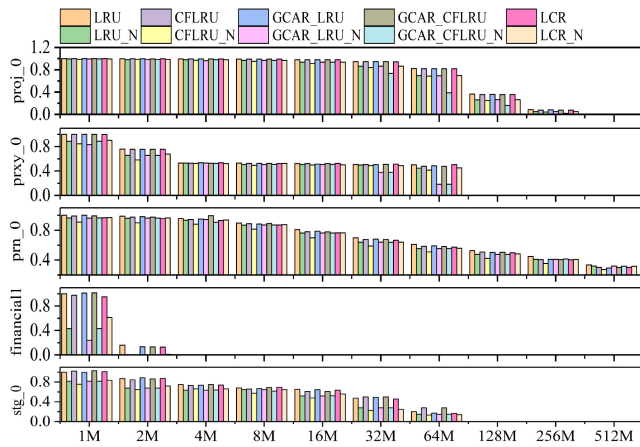


Fig. 9. y-axis represents erase count, normalized to that of the LRU scheme under each trace. The x-axis denotes cache sizes.

NCache achieves the same performance as that of the traditional caching schemes with 4-MB cache. For *prn_0* with small spatiotemporal locality, 16-MB NCache has the same performance as that with 64-MB cache. For *WebSearch_1*, the 512-MB-cache performance is similar to that of NCache with 64-MB cache. Unlike NCache, the spatiotemporal-locality caching scheme is inadequate to optimize performance under weak-locality traces.

Erase Count: Fig. 9 reveals that NCache is a winner among its peers in terms of erase count. NCache saves footprint for valid data, thereby lessening frequent I/O accesses to flash memory and cutting back the number of erasures by the valid data. Under write-intensive workloads, NCache provides noticeable improvement for erase count with the cache size increment. The erase count decreases with a large-sized cache. In the case of 2-MB cache, NCache presents no erasure by improvement 100% at best under *financial1*. With 512-MB cache, the erase count is 0 under *proj_0*. Under *prxy_0* and

stg_0, the erase count becomes zero when the cache size is 128 MB. The erase count improves by up to 62.3% and 55.3% under *prxy_0* and *stg_0*, respectively. There are always block-erase operations under *prn_0*. NCache improves erase count by up to 16.5% with an increasing cache size.

WA: Recall that WA indicates write-back operations of caching schemes to flash memory, see Fig. 10. NCache optimizes WA by up to 48% and 40.9% under *proj_0* and *prxy_0*, respectively. Under *stg_0*, *prn_0*, and *financial1*, the improvement of WA achieves up to 6.8%, 15.9%, and 13.9% with the 32-, 128-, and 256-MB cache, respectively. Under read-intensive workloads with 128-MB cache, NCache-GCaR_LRU brings down WA by 20.5% under *hm_1*. WA drops because NCache improves footprint and cuts down flash memory operations. *Websearch_1* almost has no write requests; the WA becomes one. For read-intensive workloads, NCache reserves cache space for read requests and evicts write requests into flash memory in advance. Write requests are dirty data spurring out-place updates in flash memory. The other caching schemes except LRU preferentially replaces clean data without writing flash memory; thus, WA is reduced.

Discussion: Given any large-sized cache (256 or 512 MB), NCache-based WA and erasure count have no superiority compared to the other schemes (see Figs. 9 and 10). CFLRU, GCaR, and LCR are write-back cost-oriented caching schemes that preferentially evict clean data rather than writing dirty data into flash memory, reducing WA and erase count. Combining with NCache, the caching schemes consider the write-back cost and the prediction result of NCache. Clean data are forecasted as valid data retained in cache; however, dirty data may be written back into flash memory, increasing WA and erase count. NCache-LRU stays away from these anomaly results because LRU is nonwrite-back-cost-oriented scheme, in which NCache imposes zero impact on write-back cost.

Read Hit Ratio (Read HR): In Fig. 11, NCache-based caching schemes boost *read HR* by up to 42.2%, 32.7%, 29.5%, 12.7%, and 52.9% in the 32-, 16-, 256-, and 16-MB

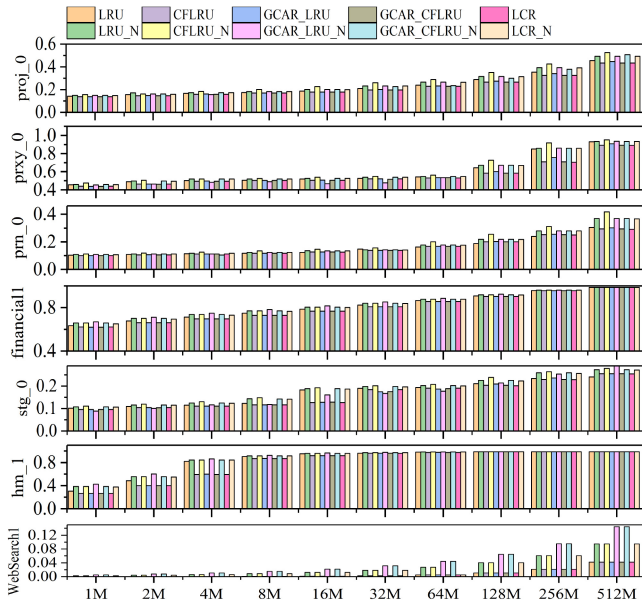


Fig. 11. y-axis denotes read *HR* under different traces and cache sizes.

cache under write-intensive traces *prn_0*, *proj_0*, *prxy_0*, *financial1*, and *stg_0*, respectively. NCache-GCaR_CFLRU boosts *read HR* by up to 59.62 times and 59.9% under *WebSearch1* and *hm_1* with the 1-MB cache.

The hit ratio is impacted by the cache size under *hm_1* with a good spatiotemporal locality. As the cache expands, the hit ratio rapidly goes up. After the hit ratio of traditional cache schemes reaches the peak at the 8-MB cache, the hit ratio is flattened. Obviously, 4-MB-cache NCache achieves the best hit ratio. Under *WebSearch1* with the weak locality, the traditional cache schemes make the same performance improvement. NCache's hit ratio in the 64-MB cache is the same as that of the traditional cache schemes in the 512-MB cache. Under write-intensive traces, the hit ratio rises as cache size is enlarged. The improvement is not stunning with the small-sized cache due to weak locality. For *stg_0*, *prxy_0*, and *proj*, the hit ratio starts to change insignificantly with the increasing cache size. In the 128-MB cache, we observe a sudden increase in the hit ratio.

Write Hit Ratio (Write HR): NCache is adept at optimizing the *write HR*, see Fig. 12. In the 64-MB cache, NCache-GCaR_CFLRU improves the *write HR* by up to a factor of 1.35 under *proj_0*. NCache-LRU boosts the metric by a factor of 66.33 under *WebSearch1*. NCache achieves high *write HR* thanks to its full cache utilization—an increased amount of valid data is served in the cache. But, the hit ratio of other combined NCache solutions declines under *WebSearch1*. NCache writes little dirty data back into flash memory, conserving space for reads and reducing the *write HR*. Except LRU, other cache schemes preferentially replace the clean data generated by reads.

The optimization degree of NCache varies with the cache size. The *write HR* is improved by 22% and 12.7% under *stg_0* and *financial1*, respectively. The improvement of NCache becomes small with a large cache size. Under *proj_0*, *prxy_0*, and *prn_0*, the optimization of NCache is pronounced as cache

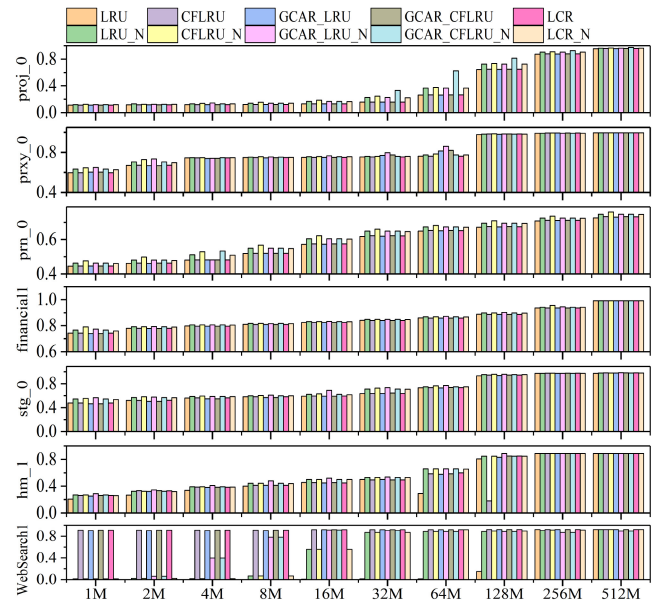


Fig. 12. y-axis denotes write *HR* under various traces and cache sizes.

size becomes large. The optimization degree reaches the peak at 64-, 2-, and 4-MB cache, respectively. The most remarkable improvement of the *write HR* is 135%, 10%, and 10.6% under *proj_0*, *prxy_0*, and *prn_0*, respectively. Afterward, as the cache size is expanded, the optimization degree of NCache is shrinking. The best optimization degree is a factor of 3.65 with the 128-MB cache under *hm_1*. NCache-LRU achieves the highest improvement of a factor of 66.3 under *WebSearch1* with the 64-MB cache because NCache gives a high priority to reads and *WebSearch1* almost has no writes.

VIII. SCALABILITY EVALUATION

We also study the benefits of NCache from two perspectives: 1) the impact of valid data ratio in memory on the cache performance and 2) the impacts of the depth parameter in decision trees on accuracy and performance.

Valid Data Ratio: Valid data ratio is the ratio of valid data to the total cached data. It denotes a caching scheme's cache space utilization. Fig. 13 shows the valid data ratios in the NCache-based caching schemes under four workloads with the 1-MB cache. The NCache-powered schemes outperform the non-NCache counterparts in terms of cache space utilization. NCache-GCaR delivers a better performance improvement than NCache-LCR. GCaR offers the most significant improvement whereas LCR provides subpar improvement. Under the read-intensive workloads, the NCache-enabled and non-NCache schemes exhibit relatively stable valid data ratios. The trend means that the spatiotemporal locality applied in the LRU-based schemes changes little.

The read-intensive workloads rarely triggers GC; thus, the cache performance is mostly determined by the hit ratio. In the above experiments, the traditional caching schemes almost have no performance discrepancy. The valid data ratio varies under the write-intensive workloads. The caching schemes take advantage of flash memory features; there is a performance gap among all the LRU-based schemes.

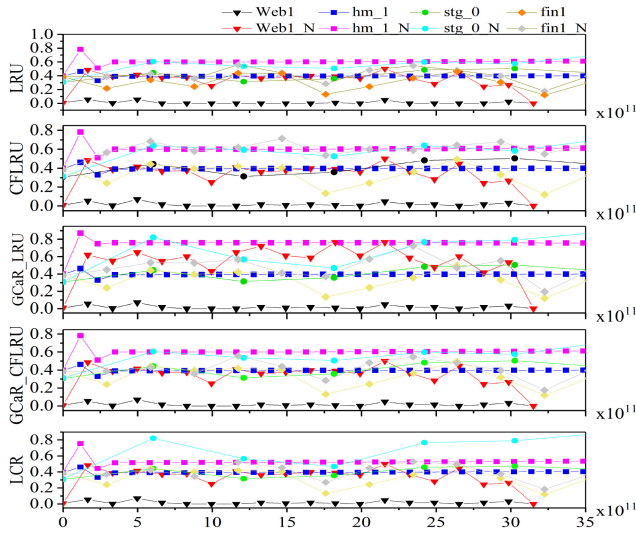


Fig. 13. Valid Data Ratio of the tested caching schemes. The x-axis represents execution time (ns), and the y-axis denotes the valid data ratio.

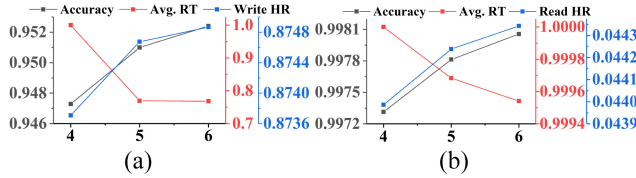


Fig. 14. Accuracy, avg. RT, and hit ratio of the model at the depth of 4, 5, and 6 in x-axis. Avg. RT is normalized to that of depth 4. (a) financial1. (b) WebSearch1.

Depth Parameters in Decision Trees: During model training, we decrease the decision-tree depth to shorten the training time and overfitting. When the tree depth under financial1 rises from four to five, the accuracy and performance are brushed up (see Fig. 14). The accuracy, performance, and hit ratio enjoy an improvement by 0.39%, 23%, and 0.1%, respectively. However, the improvement of the metrics becomes marginal when the tree depth changes from five to six. Specifically, the accuracy, performance, and hit ratio increases by 0.14%, 0.14%, and 0.022%, respectively. Such a tendency is also applicable for Websearch1. We observe that accuracy, performance, and hit are improved by 0.05%, 0.032%, 0.57%, when the tree depth rises from four to five. These three measures have a meager improvement when the depth is altered from five to six.

The results suggest that the depth of decision trees ought to be limited when features have continuous values. Amid training for decision trees, the discrete attributes that have been used do not appear again in the subsequent classification. Thus, when there are five discrete features, the tree depth is five. Nevertheless, the same attribute may appear multiple times because of different nodes in a tree for continuous variables. Without the depth limitation, the decision tree continues growing. A large depth of the decision tree leads to long training time and overfitting, which is detrimental to the improvement of accuracy, avg. RT, and hit ratio. To address this concern, we set the depth of the decision tree to five. Besides, our empirical study confirms that the depth of five offers a good tradeoff between training cost and performance measures in decision trees.

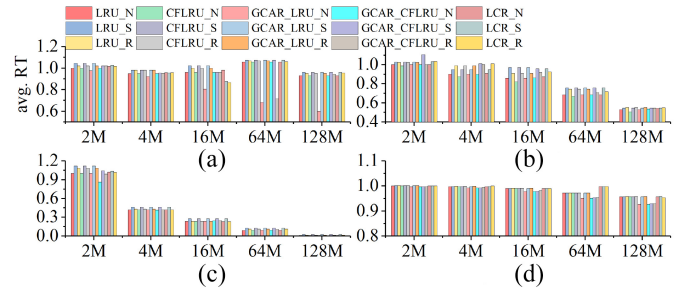


Fig. 15. Avg. RT, normalized to that of the LRU_N scheme. LRU_N means NCACHE-based LRU. LRU_S and LRU_R denote that NCACHE variants equipped with the SVM and random forest models. (a) Financial 1. (b) stg_0. (c) hm_1. (d) WebSearch1.

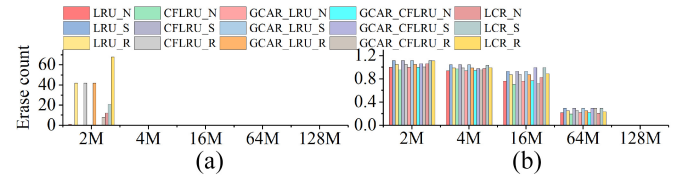


Fig. 16. Erase count, normalized to that of the LRU_N scheme. (a) Financial 1. (b) stg_0.

Performance Comparison of ML Models in NCACHE: We studied the performance of NCACHE when the SVM and random forest models are deployed in NCACHE. The two models are also used in the cache replacement procedure [28], [29]. *S-NCACHE* and *RF-NCACHE* denote the NCACHE variants equipped with the SVM and random forest models, respectively. Without loss of generality, we employ five cache sizes under write-intensive traces (financial1 and stg_0) and read-intensive traces (hm_1 and WebSearch1).

SVM is a supervised learning-based binary classification model. To classify the samples, we adopt SVM to find the classification hyperplane with the largest interval in a feature space. Given large-scale samples, it is difficult to discover the optimal classification hyperplane. There may be misclassification points, causing inaccurate classification. SVM is time and space consuming when it comes to training with large-scale samples. A training model is too large to be accommodated inside SSDs. Random forest is a classifier that uses multiple trees to predict. Each tree randomly selects a subset of features for training purposes, reducing the number of features. The five workload features have vital connections and are crucial for predicting valid data. The subset of features can downgrade prediction accuracy.

In Fig. 15, NCACHE has a lower avg. RT than the other schemes. Under hm_1, the 64-MB NCACHE achieves the improvement of 38% and 23% over *S-NCACHE* and *RF-NCACHE* because NCACHE provides a better prediction for valid data, improves cache space utilization, and averts access to flash memory. For weak spatiotemporal-locality in WebSearch1, the avg. RT values of the three schemes are similar: NCACHE has the best enhancement (3.2%) in the 128-MB cache.

Figs. 16 and 17 show that NCACHE's erase count and WA are lowest. Erase count rarely occurs under read-intensive workloads. NCACHE has no erase count under financial and stg_0 after 4 and 128 MB, respectively. SVM and random forest have lower cache space utilization than NCACHE due to

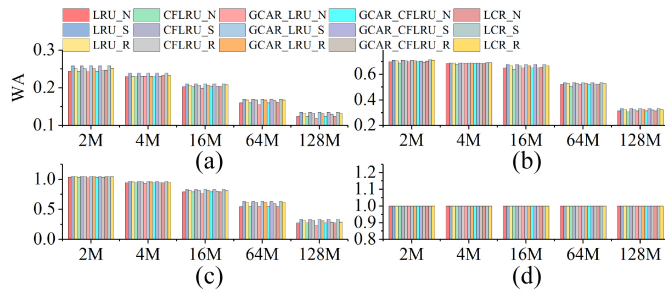


Fig. 17. WA under various traces and cache sizes. (a) Financial 1. (b) stg_0. (c) hm_1. (d) WebSearch 1.

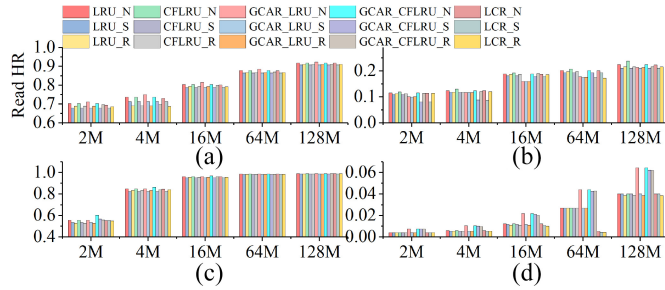


Fig. 18. Read HR under various traces and cache sizes. (a) Financial 1. (b) stg_0. (c) hm_1. (d) WebSearch 1.

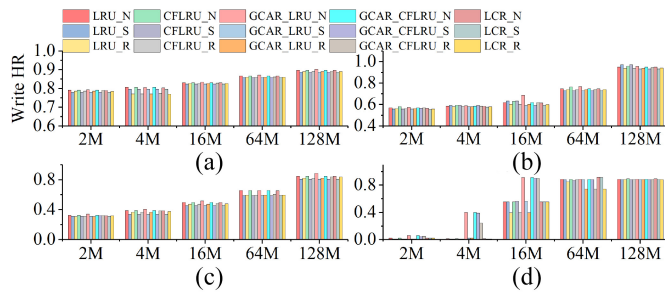


Fig. 19. Write HR under various traces and cache sizes. (a) Financial 1. (b) stg_0. (c) hm_1. (d) WebSearch 1.

inaccurately prediction for valid data; thus, there are many erase counts. The 64-MB NCache has a 39% and 23% reduction compared with SVM and random forest under stg_0 because NCache fully utilizes cache space and reduces the operation in flash memory to curtail WA. Under WebSearch1, WA is one thanks to the lack of write requests. Under hm_1 and 128-MB cache, NCache slashes WA by up to 42% and 35% compared with *S*- and RF-NCache.

In Fig. 18, NCache exhibits the highest *read HR* among the three schemes. The 4-MB NCache improves the *read HR* by up to 30% compared with *S*- and RF-NCache under stg_0, respectively. For the read-intensive traces, NCache optimizes the *read HR* by up to 47% and 50% over *S*- and RF-NCache, respectively, under WebSearch1. These improvements are made possible by NCache's high cache utilization contributed by accurate predictions. In Fig. 19, NCache achieves the highest write HR than the other alternatives. The 16-MB NCache boosts write HR by up to 14% and 13% over *S*- and RF-NCache, respectively, under stg_0. Under the read-intensive traces, the 4-MB NCache achieves 94% and 95%

advancements over *S*- and RF-NCache in terms of write HR under WebSearch1. The reason is that NCache has a high-write request accuracy leading to enormous valid data that pushes up cache space utilization.

IX. CONCLUSION AND DISCUSSION

Conclusion: In this article, we proposed a caching scheme, NCache, to take advantage of an ML model to boost hit ratio through valid data discovery. NCache expands the proportion of valid data in the cache by replacing data that are less likely to be accessed according to the ML model prediction. NCache is highly accurate because it is conducive to distinguishing data classes in cache. NCache rarely sacrifices the hit ratio to optimize the SSD performance. When I/O requests are not closely correlated, locality-based caching schemes have an arduous process to guarantee high hit ratio and cache performance. NCache overcomes this problem by fully utilizing cache's workload traits rather than historical I/O access information. For example, we compared NCache with five existing caching schemes. Under write-intensive trace pmn_0, the NCache-based schemes shorten *avg. RT* by up to 15.0%. Besides, under read-intensive trace hm_1, the NCache-based schemes significantly cut back *avg. RT* by up to 56.7% with an average of 27.5%.

Discussion: In this study, we opted for a binary classification model instead of a multiclass classification model to classify cache data into two camps—1) valid data (Class-1, 1) and 2) invalid data (Class-1, 0). For the multiclass classification model, the 5-level decision tree may not fulfill the prediction-accuracy requirements. To grapple with this challenge, we enlarged the decision tree's depth to achieve the stipulated accuracy, but this approach eventually introduces more classification conditions, making the model's time overhead tool high to be executed inside SSDs. We sorted in NCache data with the same class in a linked list, aiming to realize various data priorities in the same class. This method achieves the multiple-class function offered by a multiclass classification model. Recall that LinkL1 registers all cache data, and LinkL2 merely stores invalid data predicted by the ML model. The tail node in LinkL2 is preferentially replaced due to the lowest priority. Conversely, when all cache data are categorized as Class-1, LinkL1 manages these data, and the tail node also will be evicted from the cache.

REFERENCES

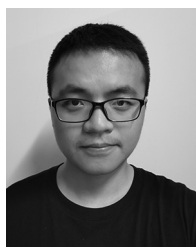
- [1] J. Wan, W. Wu, L. Zhan, Q. Yang, X. Qu, and C. Xie, "DEFT-cache: A cost-effective and highly reliable SSD cache for RAID storage," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 102–111.
- [2] Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A flexible, high-performance key-value SSD," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2017, pp. 373–384.
- [3] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *Proc. USENIX Annu. Tech. Conf.*, 2008, pp. 1–14.
- [4] S.-Y. Park, D. Jung, J.-U. Kang, J.-S. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proc. Int. Conf. Compilers Archit. Synth. Embedded Syst.*, 2006, pp. 234–241.
- [5] C. Liu et al., "LCR: Load-aware cache replacement algorithm for flash-based SSDs," in *Proc. IEEE Int. Conf. Netw. Archit. Storage*, 2018, pp. 1–10.
- [6] S. Wu, Y. Lin, B. Mao, and H. Jiang, "GCAR: Garbage collection aware cache management with improved performance for flash-based SSDs," in *Proc. Int. Conf. Supercomput.*, 2016, pp. 1–12.

- [7] S. Yan *et al.*, “Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs,” *ACM Trans. Storage*, vol. 13, no. 3, pp. 1–26, 2017.
- [8] J. Guo, Y. Hu, B. Mao, and S. Wu, “Parallelism and garbage collection aware I/O scheduler with improved SSD performance,” in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2017, pp. 1184–1193.
- [9] W. W. Hsu and A. J. Smith, “The performance impact of I/O optimizations and disk improvements,” *IBM J. Res. Develop.*, vol. 48, no. 2, pp. 255–289, Mar. 2004.
- [10] A. Dan and D. Towsley, “An approximate analysis of the LRU and FIFO buffer replacement schemes,” in *Proc. ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.*, 1990, pp. 143–152.
- [11] Y. Smaragdakis, S. Kaplan, and P. Wilson, “EELRU: Simple and effective adaptive page replacement,” *SIGMETRICS Perform. Eval. Rev.*, vol. 27, no. 1, pp. 122–133, 1999.
- [12] T. Johnson and D. Shasha, “2Q: A low overhead high performance buffer management replacement algorithm,” in *Proc. 20th Int. Conf. Very Large Data Bases*, 1994, pp. 439–450.
- [13] D. Lee *et al.*, “LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies,” *IEEE Trans. Comput.*, vol. 50, no. 12, pp. 1352–1361, Dec. 2001.
- [14] N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache,” in *Proc. USENIX Conf. File Storage Technol.*, 2003, pp. 115–130.
- [15] D. Lee *et al.*, “On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies,” *SIGMETRICS Perform. Eval. Rev.*, vol. 27, no. 1, pp. 134–143, 1999.
- [16] R. Karedla and J. S. Love, “Caching strategies to improve disk system performance,” *Computer*, vol. 27, no. 3, pp. 38–46, 1994.
- [17] A. Jain and C. Lin, “Back to the future: Leveraging Belady’s algorithm for improved cache replacement,” in *Proc. 43rd Int. Symp. Comput. Archit.*, 2016, pp. 78–89.
- [18] S. Das, T. M. Aamodt, and W. J. Dally, “Reuse distance-based probabilistic cache replacement,” *ACM Trans. Archit. Code Optim.*, vol. 12, no. 4, pp. 1–22, 2015.
- [19] M. Jung, W. Choi, S. Srikantaiah, J. Yoo, and M. T. Kandemir, “HIOS: A host interface I/O scheduler for solid state disks,” *ACM SIGARCH Comput. Archit. News*, vol. 42, no. 3, pp. 289–300, 2014.
- [20] X. Zhang, J. Li, H. Wang, K. Zhao, and T. Zhang, “Reducing solid-state storage device write stress through opportunistic in-place delta compression,” in *Proc. USENIX Conf. File Storage Technol.*, 2016, pp. 111–124.
- [21] J. Hu, H. Jiang, L. Tian, and L. Xu, “GC-ARM: Garbage collection-aware RAM management for flash based solid state drives,” in *Proc. IEEE Int. Conf. Netw. Archit. Storage*, 2012, pp. 134–143.
- [22] H. Kim and S. Ahn, “BPLRU: A buffer management scheme for improving random writes in flash storage,” in *Proc. USENIX Conf. File Storage Technol.*, 2008, pp. 239–252.
- [23] H. Chen, Y. Pan, C. Li, and Y. Xu, “ECR: Eviction-cost-aware cache management policy for page-level flash-based SSDs,” *Concurr. Comput.*, vol. 33, no. 15, p. e5395, 2021.
- [24] H. Jo, J.-U. Kang, S.-Y. Park, J.-S. Kim, and J. Lee, “FAB: Flash-aware buffer management policy for portable media players,” *IEEE Trans. Consum. Electron.*, vol. 52, no. 2, pp. 485–493, May 2006.
- [25] S. Liang, Y. Wang, Y. Lu, Z. Yang, H. Li, and X. Li, “Cognitive SSD: A deep learning engine for in-storage data retrieval,” in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 395–410.
- [26] Z. Deng, L. Zhang, N. Mishra, H. Hoffmann, and F. T. Chong, “Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in NVMs,” in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 232–244.
- [27] D. A. Jiménez and E. Teran, “Multiperspective reuse prediction,” in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2017, pp. 436–448.
- [28] Z. Shi, X. Huang, A. Jain, and C. Lin, “Applying deep learning to the cache replacement problem,” in *Proc. IEEE/ACM Int. Symp. Microarchit.*, 2019, pp. 413–425.
- [29] K. Vaithinathan, J. B. Pernabas, L. Parthiban, B. Shrestha, G. P. Joshi, and H. Moon, “An improved Web caching system with locally normalized user intervals,” *IEEE Access*, vol. 9, pp. 112490–112501, 2021.
- [30] H. Wang, X. Yi, P. Huang, B. Cheng, and K. Zhou, “Efficient SSD caching by avoiding unnecessary writes using machine learning,” in *Proc. Int. Conf. Parallel Process.*, 2018, pp. 1–10.
- [31] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.
- [32] G. Keramidas, P. Petoumenos, and S. Kaxiras, “Cache replacement based on reuse-distance prediction,” in *Proc. Int. Conf. Comput. Des.*, 2007, pp. 245–250.
- [33] M. Kharbutli and Y. Solihin, “Counter-based cache replacement and bypassing algorithms,” *IEEE Trans. Comput.*, vol. 57, no. 4, pp. 433–447, Apr. 2008.
- [34] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, “Semantic locality and context-based prefetching using reinforcement learning,” in *Proc. ACM/IEEE Int. Symp. Comput. Archit.*, 2015, pp. 285–297.
- [35] B. Li, J. Friedman, R. Olshen, and C. Stone, “Classification and regression trees (CART),” *Biometrics*, vol. 40, no. 3, pp. 358–361, 1984.
- [36] E. Apaydin, *Introduction to Machine Learning*, 4th ed. Cambridge, MA, USA: MIT Press, 2020.
- [37] G. Baryannis, S. Dani, and G. Antoniou, “Predicting supply chain risks using machine learning: The trade-off between performance and interpretability,” *Future Gener. Comput. Syst.*, vol. 101, pp. 993–1004, Dec. 2019.
- [38] J. R. Quinlan, “Induction of decision trees,” *Mach. Learn.*, vol. 1, no. 1, pp. 81–106, 1986.
- [39] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA, USA: Morgan Kaufmann Publ., 1993.



Hui Sun received the Ph.D. degree from Huazhong University Science and Technology, Wuhan, China, in 2014.

He is an Associate Professor of Computer Science with Anhui University, Hefei, China. His research interests include computer systems, performance evaluation, storage systems, and I/O architectures.



Qiao Cui is currently pursuing the M.S. degree with Anhui University, Hefei, China.

His main research interests include computer systems, SSD storage, and intelligent cache management.



Jianzhong Huang (Member, IEEE) received the Ph.D. degree in computer architecture from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2005.

He was a Postdoctoral Researcher of Information Engineering with HUST in 2007, where he is currently an Associate Professor with the Wuhan National Laboratory for Optoelectronics. His research interests include computer architecture and dependable storage systems.



Xiao Qin (Senior Member, IEEE) received the B.S. and M.S. degrees in computer science from Huazhong University of Science and Technology, Wuhan, China, in 1992 and 1999, respectively, and the Ph.D. degree in computer science from the University of Nebraska–Lincoln, Lincoln, NE, USA, in 2004.

He is currently an Alumni Professor and the Director of Graduate Programs with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL, USA. His research

interests include parallel and distributed systems, storage systems, and performance evaluation.

Dr. Qin won the NSF CAREER Award in 2009.