

A Machine Learning-Empowered Cache Management Scheme for High-Performance SSDs

Hui Sun , Chen Sun , Haoqiang Tong , Yinliang Yue , and Xiao Qin , *Senior Member, IEEE*

Abstract—NAND Flash-based solid-state drives (SSDs) have gained widespread usage in data storage thanks to their exceptional performance and low power consumption. The computational capability of SSDs has been elevated to tackle complex algorithms. Inside an SSD, a DRAM cache for frequently accessed requests reduces response time and write amplification (WA), thereby improving SSD performance and lifetime. Existing caching schemes, based on temporal locality, overlook its variations, which potentially reduces cache hit rates. Some caching schemes bolster performance via flash-aware techniques but at the expense of the cache hit rate. To address these issues, we propose a random forest machine learning Classifier-empowered Cache scheme named CCache, where I/O requests are classified into critical, intermediate, and non-critical ones according to their access status. After designing a machine learning model to predict these three types of requests, we implement a trie-level linked list to manage the cache placement and replacement. CCache safeguards critical requests for cache service to the greatest extent, while granting the highest priority to evicting request accessed by non-critical requests. CCache – considering chip state when processing non-critical requests – is implemented in an SSD simulator (SSDSim). CCache outperforms the alternative caching schemes, including LRU, CFLRU, LCR, NCache, ML_WP, and CCache_ANN, in terms of response time, WA, erase count, and hit ratio. The performance discrepancy between CCache and the OPT scheme is marginal. For example, CCache reduces the response time of the competitors by up to 41.9% with an average of 16.1%. CCache slashes erase counts by a maximum of 67.4%, with an average of 21.3%. The performance gap between CCache and OPT is merely 2.0%-3.0%.

Index Terms—Cache management schemes, machine Learning, random forest classifier, NAND flash, solid state disks.

I. INTRODUCTION

NAND Flash-based SSDs, being widely deployed in storage systems [1], offer prominent performance, low power consumption, and enhanced reliability compared to traditional

HDDs [2]. A portion of DRAM [3], [4] inside an SSD is employed as a cache, the purpose of which is to act as a performance bridge between the host-side I/O interface and NAND flash memory [5]. Cache temporarily stores frequently read and write requests. When a request matches the same logical page address (LPA) within the cache, the request can be swiftly served – reducing the request's response time and minimizing the number of reads and writes performed on the flash memory. Meanwhile, improvements are observed in the performance and lifetime of SSDs [6]. Cache space is limited due to high cost. Thus, an efficient cache scheme, which optimizes the utilization of the cache space, plays an essential role in revamping the performance and lifetime of SSDs.

To improve SSD performance, existing cache schemes focus on elevating the cache hit rate by exploring temporal locality in workloads such as LRU [7] – a basic cache scheme. CFLRU [8], LCR [9], and GCaR-LRU [10] are extensions derived from LRU. CFLRU with temporal locality offers a mechanism of prioritizing filtering clean requests to be evicted, thereby avoiding the extra write-backs caused by replacing dirty requests from the cache. LCR replaces request according to the lengths of chip queues, and GCaR considers garbage collection (GC) status in flash chips. Incorporating both I/O access patterns and flash memory status, DAC [11] and Co-Active [12] are adaptive to changes in I/O patterns while fully utilizing information such as chip queue and GC state to handle cache requests. These cache schemes depend on the temporal locality of workloads. If the workloads are featured with a solid temporal locality, a locality-based caching scheme presents a high cache hit rate, improving SSD performance. Otherwise, these cache schemes perform poorly under workloads with a weak temporal locality. To address this issue, NCache [13] with a machine learning (ML) model pre-predicts valid and invalid requests based on I/O features to improve cache hit rate and optimize cache performance. Wang et al. [14] raises cache utilization by writing one-time access requests to flash memory to bypass cache. However, this study considers the overall temporal locality in the workload, where the temporal locality across multiple stages is dissimilar, leading to cold requests occupying the space of hot requests in the cache.

We propose a random forest classifier-empowered cache scheme named CCache. CCache solves the problem of dynamically changing temporal locality in workloads, while traditional temporal locality-based cache schemes relying on temporal locality rarely provide an efficient means to reshape SSD performance. Initially, during the offline stage, CCache

Manuscript received 30 September 2023; revised 30 March 2024; accepted 5 May 2024. Date of publication 22 May 2024; date of current version 11 July 2024. This work was supported by the National Natural Science Foundation of China under Grant 61702001. 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. Recommended for acceptance by Y. Chen. (Corresponding author: Yinliang Yue.)

Hui Sun, Chen Sun, and Haoqiang Tong are with the School of Computer Science and Technology, Anhui University, Hefei 230201, China (e-mail: sunhui@ahu.edu.cn; e21301220@stu.ahu.edu.cn; e21301287@stu.ahu.edu.cn).

Yinliang Yue is with the Zhongguancun Laboratory, Beijing 100049, China (e-mail: yylhust@qq.com).

Xiao Qin is with the Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849 USA (e-mail: xqin@auburn.edu).

Digital Object Identifier 10.1109/TC.2024.3404064

obtains the lifetime of requests in the cache by applying our proposed Labeler algorithm (see §IV). Then, we build and train a machine-learning model: The features gleaned by CCache's feature collector are taken as input information for the model, and CCache acquires labels – critical request, intermediate request, and non-critical request. Notably, requests that will be frequently accessed in the foreseeable future are labeled as critical requests; requests that will be rarely accessed are marked as non-critical requests; intermediate requests for occasionally accessed requests are tagged to smooth out the critical and non-critical requests. CCache selectively caches non-critical requests according to the chip state. CCache maximally caches critical requests so that they can be consistently served by the cache. Non-critical requests are evicted prior to the other two-type requests during cache replacement. Our contributions are listed as follows.

△ We propose a random forest classifier-empowered cache scheme – CCache – to accurately grasp I/O access patterns in the context of varying temporal locality [15] in workloads, where the system performance is promoted.

△ We advocate for ML to classify I/O requests into three camps, namely, critical, intermediate, and non-critical. Critical requests are given priority to be serviced, whereas request handled by non-critical requests are evicted in the first place. A tri-level linked list is formulated to expedite the organization of I/O requests.

△ We propose a degradation mechanism for I/O requests to prevent critical request from monopolizing cache space over extended periods. This mechanism also averts an empty linked list of non-critical requests - a scenario that may raise due to the intricacies of ML accuracy.

△ After implementing CCache in SSDsim, we quantitatively compare CCache with the seven popular cache schemes. For example, CCache curtails the response time by a maximum of 41.9% with an average of 16.1%. CCache reduces erase counts by up to 67.4% with an average of 21.3%. The performance gap between CCache and OPT – the theoretically optimal algorithm – is anywhere between 2% and 3%.

The remainder of this paper is organized as follows. We introduce the background in Section II. We discuss the motivations of this study in Section III. The CCache design is articulated in Section IV. In Section V, we elaborate on the performance of CCache compared with the alternative caching schemes. We present related work in Section VI. Finally, we conclude our work.

II. BACKGROUND

SSDs offer superior performance and reduced power consumption, compared with HDDs. Fig. 1 depicts the structure of an SSD, which is composed of host interface logic (HIL), back-end flash storage, and a flash translation layer (FTL) [16]. HIL connects the SSD to a host: upon the arrival of an I/O request in the host, HIL slices the request into page-sized requests. The back-end storage consists of NAND Flash, providing high parallel processing capacity that embraces four-level parallelism – channel, chip, plane, and die. In flash memory, a block is the basic erasure unit, and a page is a unit for reads and

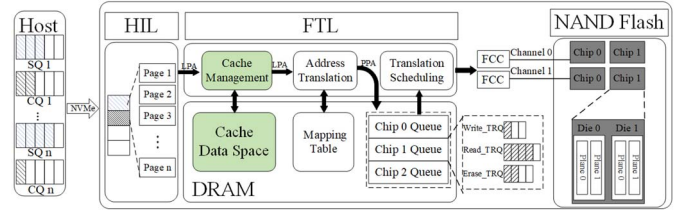


Fig. 1. The architecture of an SSD. SQ: I/O submission queue. CQ: I/O completion queue.

writes. FTL handles cache management, address translation, garbage collection, and wear-leveling [17], [18]. An SSD applies the ECC method to render high reliability in flash memory [19]. A portion of DRAM is set to a cache that temporarily stores requests. The cache alleviates any mismatch of transfer performance between the host and the flash memory. When the host sends a read request, the SSD can pre-read data from the flash memory into the cache. Thus, when the application requests the same data, the SSD can read the data from the cache rather than access the flash memory: read latency is shortened. Host-side write data are placed in the cache before writing the data to the flash memory. The cache can merge multiple writes, reducing the number of writes to the flash memory and improving the SSD lifetime.

Machine learning algorithms are slated to collect and analyze request features, which are trained to learn intrinsic from requests. Deploying ML techniques in the cache management is viable thanks to the advancement of the computational power of SSDs. For instance, SSDKeeper [20] adopts a channel allocation mechanism based on the ANN model, which analyzes I/O access patterns in workloads before allocating channels. In MLCache [21], the reuse distance of requests are calculated – and ANN is employed to allocate cache space for requests in a multi-stream environment. NCache facilitates cache space management by providing distinct decision results for requests. ML_WP [22] proposes a ML model based write policy that reduces write traffic to the SSD by avoiding write-only data. The challenge of this approach is to recognize write-only data in real time.

III. MOTIVATION

The existing LRU-based cache schemes are well adapted to workloads with intensely temporal locality, where recently accessed data will be handled again in the not-too-distant future. However, the temporal locality in a workload rarely remains at a high level; thus, these cache schemes struggle to deliver satisfactory performance in such scenarios.

We delve into the temporal locality using the three realistic traces at Microsoft Research Cambridge (MSR) [24] and five workloads from Systor'17 Traces [25], see Table I. Total_Locality and Segment_Locality represent the overall temporal locality and a segment temporal locality in a workload. For the Segment_Locality, we take 500 I/Os as a period and divide the workload into several sets of stages ($A_1, A_2, \dots, A_i$). A_i signifies the set of LPAs for the i^{th} period. The intersection

TABLE I
DETAILS OF TRACES USED IN THIS PAPER

Trace ¹	Total I/Os	Write Size (GB)	Read Size (GB)	Average Size (KB)	Loc.T ²	Loc.SD ³	Cr.T ⁴	Cr.SD ⁵	R ² ⁶
mds_0	1048576	6.31	3.19	9.51	0.55	0.28	0.522	0.31	0.99
stg_0	2030915	15.09	7.33	11.58	0.430	0.20	0.41	0.23	0.96
ts_0	1801734	11.33	4.13	9.00	0.53	0.19	0.49	0.24	0.97
LUN2-1	4443472	24.14	19.74	10.35	0.42	0.24	0.51	0.23	0.83
LUN2-2	3756891	19.92	23.97	12.25	0.40	0.24	0.41	0.29	0.83
LUN2-3	2902315	13.89	16.29	10.91	0.46	0.30	0.49	0.32	0.89
LUN2-4	2408200	13.48	8.08	9.39	0.54	0.29	0.56	0.32	0.88
LUN0-1	1997972	6.48	51.92	30.65	0.19	0.23	0.12	0.25	0.97

¹Symbols LUN2-1, LUN2-2, LUN2-3, LUN2-4, and LUN0-1 in column 1 represent traces 2016030817-LUN2, 2016030916-LUN2, 2016030819-LUN2, 2016031115-LUN2, and 2016031816-LUN0, respectively. ²Loc.T is Total_Locality of workload. ³Loc.SD is the standard deviation of Segment_Locality in the workload. ⁴Cr.T is Total_Critical_Req of workload. ⁵Cr.SD is the standard deviation of Segment_Critical_Req of stages in the workload. ⁶R² [23] represents the correlation coefficient. R² indicates the degree to which Segment_Locality and Segment_Critical_Req are linearly correlated, with larger values indicating greater correlation.

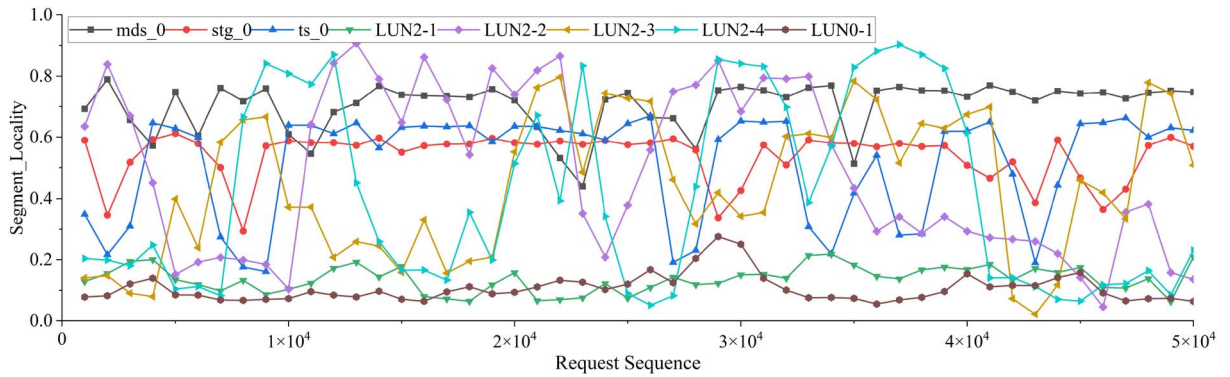


Fig. 2. The trend of temporal locality in each stage of the workload. X-axis represents the request sequence in workloads.

of A_i and A_{i+1} is N_i is given as

$$\begin{aligned}
 N_1 &= A_1 \cap A_2 \\
 N_2 &= A_2 \cap A_3 \\
 &\dots \\
 N_i &= A_i \cap A_{i+1}
 \end{aligned} \tag{1}$$

$|N_i|$ denotes the number of sets N_i , and the ratio of $|N_i|$ to 500 is the temporal locality (*Segment_Locality*) for the i^{th} period. The total locality of the workload, *Total_Locality*, is defined as $Total_Locality = (|N_1| + |N_2| + \dots + |N_i|) / (i \times 500)$.

Fig. 2 sketches the variation of *Segment_Locality* for the first fifty time periods of the eight workloads. We reckon that the *Segment_Locality* of each stage varies under the workload, and stage temporal locality may present fluctuation in some workloads. In *mds_0*, the *Segment_Locality* is a minimum of about 0.44, and the maximum can reach 0.79. Compared with *mds_0*, LUN2-4 shows a greater fluctuation, which indicates that temporal locality is constantly changing. The *Segment_Locality* of LUN2-1 and LUN2-2 exhibit a high level of smoothness that is in a lower range of values. *Total_Locality* in Table I is the overall locality of a workload, with a value of 0.55 for *mds_0* and 0.19 for LUN0-1. *Loc.SD* unveils the degree of stage fluctuation of *Segment_Locality*. Table I suggests that a portion of workloads with high *Total_Locality* have a relatively large *Loc.SD*. For example, the *Total_Locality* of LUN2-4 achieves a value of 0.54, whereas the corresponding standard deviation reaches 0.28. This result entails many periods having relatively low segment locality even though the *Total_Locality*

is high. Hence, managing workloads based on overall locality becomes challenging.

The temporal locality of workload varies over time; at one stage, with suitable temporal locality, LRU-based caching schemes can achieve superb performance – but at other stages, there may be poor temporal locality that deteriorates the performance. Designing a cache scheme solely according to an overall temporal locality of workloads falls short of effectively optimizing performance. We analyze the impact of changes between high and low *Segment_Locality*. If a request accesses the same LPA more than twice, we define the request as a critical request. The request that retrieves an LPA once is denoted as a non-critical request. The remaining requests – being a buffer between the critical and non-critical requests – are referred to as intermediate requests: we introduce intermediate requests to avoid cliffhanging changes between critical and non-critical ones. The three-type requests increase the request granularity and achieve fine-grained request management.

Fig. 3 shows a cache replacement flow, where the cache size is 8 pages and LRU is deployed. Stage ① represents the initial distribution of different requests. In stage ②, non-critical requests occupy the space that would otherwise be allocated to critical requests, resulting in the replacement of critical requests. In the case of issuing these critical requests that lead to cache misses, performance degradation becomes inevitable. These requests are discretely distributed in the cache due to variations critical requests between two successive stages. We count the critical requests in MSR and Sys-¹⁷ traces at each stage. A critical request accesses the same LPA more than twice in the 500-I/Os stages, and we gauge

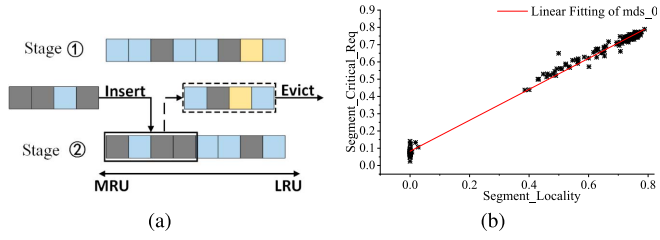


Fig. 3. (a) The different number of critical requests over two time periods. These blue, black, and yellow boxes represent critical, non-critical, and intermediate requests, respectively. (b) The scatterplot of Segment_Critical_Req (y-axis) and Segment_Locality (x-axis). The red line shows the regression line.

the critical requests in each period containing 500 I/Os. Segment_Critical_Req indicates the ratio of critical requests to 500 ones in a period. Total_Critical_Req defines the mean value of Segment_Critical_Req for each stage. In Table I, LUN2-4 has the highest Total_Critical_Req of (0.56), and LUN0-1 has the lowest Total_Critical_Req (0.13). The large *Cr.SD* implies more discrete caching of the critical requests. LUN2-4, which exhibits the highest Total_Critical_Req, displays the greatest *Cr.SD* of 0.32.

We investigate the correlation between Segment_Locality and Segment_Critical_Req. Symbol R2 in Table I denotes the correlation coefficient of the regression line of Segment_Locality and Segment_Critical_Req. There is a robust positive means that Segment_Critical_Req varies with Segment_Locality; meanwhile, the democratization correlation link between the two metrics: the high value presents a strong correlation. In Fig. 3, mds_0 is close to the regression line, and the correlation coefficient reaches 0.99. The discreteness of critical requests varies with Segment_Locality. Given the above scenario, LRU efficiently processes workloads with small and robust temporal locality at the stage level. Unfortunately, inadequate performance occurs when there are Segment_Locality fluctuations. This is attributed to varying numbers of critical requests and non-critical requests between two consecutive stages, which consumes space allocated for critical requests. Meanwhile, in the continuous stage-level temporal locality, the request types in the cache-linked list are complex, which introduces overhead in cache management.

To address the aforementioned challenges, we propose CCache – a cache scheme that adapts to the changes of Segment_Locality by judiciously handling critical, intermediate, and non-critical requests. The classification of the three requests is based on the relevance to future I/O accesses; thus, we use an ML model – random forest classifier – to categorize the request types. The ML-empowered CCache identifies critical requests by analyzing the characteristics of the requests, which are classified by the ML model into critical, intermediate, and non-critical camps maintained in three linked lists. CCache aims to manage different-typed requests in a dynamically changing Temporal_Locality. CCache stores data accessed by critical requests in the cache for an extended period. Based on the chip state, CCache conditionally writes back data of non-critical requests to flash memory. Critical data are accessed

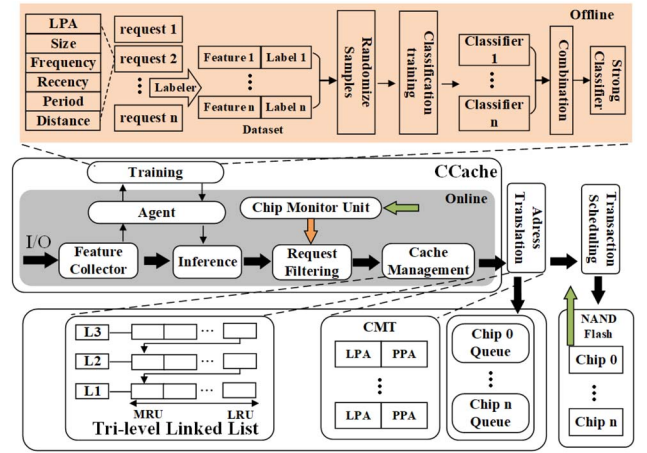


Fig. 4. Overview system of CCache

frequently, and non-critical data are less likely to be retrieved in the foreseeable future. CCache enhances both cache hit rate and performance by evicting data acquired by non-critical requests.

IV. DESIGN

A. System Overview

As shown in Fig. 4, CCache comprises the feature collection, offline training, inference, and cache management modules. CCache is deployed in the FTL inside an SSD, providing sufficient computing resources to run the ML module.

Initially, an SSD receives requests from a host and slices the requests into page-sized sub-requests through the HIL. The feature collection module harvests the features of the sub-requests in a real-time manner, rendering features to the training module via an agent. The offline training module, offering an offline process, does not impact the SSD performance. When updating the ML model, the new model is transmitted from the host to the device through the agent module. The inference module acquires the requested features from the collector, and CCache performs inference to obtain the label according to the ML model. The cache management module handles the requests. During the offline stage, the Labeler algorithm is applied to obtain the labeled dataset for training. In this study, we implement the random forest algorithm to train the dataset for a targeted ML model.

B. Feature Collector

To identify critical pages served by the cache, CCache needs to glean the features of requests in the workload; therefore, we designed the feature collector to gather the features of requests for training the ML model to obtain critical requests.

1) *Features for ML Model in CCache*: The HIL module divides host-side requests into page-sized sub-requests. The feature collection module extracts the feature information for these sub-requests. This information serves two purposes: (1) the information is delivered to the host to construct a training dataset, and (2) the information offers feature input in the inference

stage. The selection of features impacts the prediction of critical requests, which is crucial for enhancing SSD performance. In this study, we explore five features in the context of ML: *size*, *frequency*, *recency*, *period*, and *distance*, which strongly correlate to critical requests. These features are the temporal locality of the workloads. Data accessed frequently in a short period, known as *hot data*, can be retained in the cache to benefit from the temporal locality, enhancing the cache hit rate, such as LRU and LFU. Spatial locality, on the other hand, refers to the likelihood of data adjacent to the currently accessed data being re-accessed within a spatial range. The spatial locality is exploited to process these consecutive-addresses data, thereby reducing operations in flash memory, such as FAB [26] and Sp.clock [27]. FAB enhances performance by exploiting spatial locality in media data and using block-granularity cache management. but, it may be limited for non-media data and non-sequential accesses due to the lack of sequential physical addresses. Sp.clock improves write performance by evicting requests with sequential LPAs to flash memory. However, it overlooks the asymmetric read-write features of flash memory, leading to conflicts with flash I/Os. Moreover, the sequential writes may compromise temporal locality and affect the cache hit rate. In addition, the variability of physical addresses over time, accompanied by the uncertainty, leads to the underutilization of spatial locality [22].

In this paper, we focused on temporal locality instead of spatial locality in workloads: this design decision is motivated by the workload diversity and the data-block hotness through the temporal locality, which is highly relevant to our study. In Section V-C5, we study the impact of spatial locality on the copy operation during GC. In addition, these five features are highly correlated with the hotness of requests, and the labels affect the cache hit rate. Each feature is significant in achieving optimal accuracy. Please refer to Section V-C6.

Size. The request size affects cache efficiency. Given a request volume, a small-sized I/O request leads to a large number of requests. Then, the data in the cache is updated frequently. In addition, I/O requests entering the SSD with long sequences are divided into sub-requests with incremental LPAs. Data accessed by these sub-requests occupy a large-sized cache space. When a new sequence of requests arrives, the locality is poor – numerous requests are facing cache misses. Small-sized I/O writes take less time, lessening the write overhead and reducing the impact of writes on the read performance of SSDs. Therefore, small-sized requests are given high priority.

Frequency. This metric, measuring the number of accesses to the same LPA request over a period, represents the hot and cold features of a request over the past period. A small number of accesses to the LPA means the request is cold; otherwise, the request is hot. A hot request with high-frequency access indicates that the request is likely to be accessed multiple times. Thus, the cache is reserved for hot requests to diminish the number of repeated accesses to the flash memory. The cache space can be significantly utilized to reduce the cache space occupied by cold data, thereby bolstering the cache hit ratio and SSD performance.

Recency. This metric gauges the number of requests between two consecutive accesses to the same LPA. When the number of interval requests is no bigger than the cache size, the request is re-accessed after the first access hit. The request can be re-accessed in the future thanks to temporal locality. When there are many requests between two consecutive accesses to the request with the data volume of these interval requests being larger than the cache size, the requested data will be evicted as it reaches the cache the next time.

Period. Frequency does not well reflect the temporal locality, and recency only provides information about the last two accesses, which cannot be traced back very far. If there are high-frequency requests in the cache, it is infeasible to evict requests based solely on access frequency when the cache replacement occurs. The metric period can address this issue, and it is defined as the ratio of the number of requests between the first access and the latest access to the same LPA to the frequency of the LPA in a duration. A small period unveils that the request's average interval for each access is smaller, and the frequency is larger. A larger period means the request's average interval for each access is larger, and the frequency is smaller. The cache should serve small-period requests while evicting large-period ones to bolster the hit rate.

Distance. It is the number of requests for a unique LPA between two accesses to the same LPA. There are other LPAs for two consecutive accesses to the same LPA. These LPAs may be accessed multiple times rather than once in this interval. We count these multi-access requests only once in the distance calculation. Because of the limited cache space, the data may be evicted from the cache before it is accessed again. When the number of unique LPAs between two accesses is not larger than the cache size, the request will encounter a hit; otherwise, the data is evicted from the cache to spark a cache miss. The cache tends to store data that are accessed by requests with small distance values.

The HIL pre-processes the I/O requests. If the request size exceeds the page size, the request is partitioned into page-sized sub-requests. We declare a counter [13] that adds one once a sub-request is processed. We redesign the cache entries by adding four data fields – number, order, firstorder, and label. The item number is the amount of accesses to the current page. We set the number to 1 if the subrequest misses the cache; otherwise, we add one to the value. Metric order suggests the serial number of a request since it was last accessed. We use firstorder to imply the serial number of the request that first enters the cache, and its value remains unchanged from entering the cache to when the data is replaced. The values of order and firstorder can be provided by counter. The label refers to the result obtained from ML inference, which reveals the request type.

2) *Feature Computation:* We utilize the information of cache items and counters for feature computation.

We obtain the request size based on the current request. Frequency is $(N_i + 1)$ if the request LPA hits; otherwise, frequency is set to one. Table II lists notation and description used in this paper. For the recency, the current subrequest gets the access-order O_r from the counter, and recency is $(O_i - O_r)$ if the

TABLE II
NOTATIONS AND DESCRIPTION

Notation	Description
i	distance of cache node i from the linked-list header.
O_i	order: the serial number of the request that last accesses cache node i .
O_{fi}	firstorder: the serial number of the request that firstly accesses cache node i .
O_r	the serial number of the current request.
N_i	the number of hits on cache node i .
N_{Max}	the default maximum of recency, period, and distance.

request LPA hits the cache. If the request LPA misses the cache, the N_Max is set to the default value. For the period, when a new sub-request arrives, the order value of this request O_r is obtained from the counter. If the request's LPA hits the cache, the period value is $\frac{O_r - O_{fi}}{N_i + 1}$. If the cache misses, Max is an initial value of the period. For the distance, in case of an LPA is hit, the distance is the number of cache items from the current access request to the head of the linked list: the LPAs of all cache items between the head of the linked list and the current access request are unique. That is, the distance is i , and the distance is N_{Max} if there is a cache miss.

Discussion. It is of paramount importance to curtail the overhead of performing feature collection to optimize CCache performance. Let us address the space overhead issue. We add four data fields in each cache item, namely, number, order, firstorder, and label. The data type of the first three fields is an integer, and the label field uses two bits to represent the three types of ML labels – critical, intermediate, and non-critical requests. The size of an integer type is 4 B, and the size of the two bits is two bits. Then, the size of a cache item increases by 98 bits. For example, the cache size is 1 MB, the page size is 16 KB, and we have 64 cache items. The total additional space overhead is 784 Bytes. Compared with a 1-MB cache, this space overload introduced by CCache is negligible. Now, we assess the computational overhead induced by feature collection. While processing an incoming sub-request, CCache searches a cache item according to the request's LPA. Meanwhile, the data fields in the cache item are acquired. We obtain features according to the values in these data fields; thus, we attain the features while determining whether the sub-request hits the cache through simple calculations, the overhead of which is marginal.

C. Model Selection

Recall that CCache intends to cache critical requests. It is arduous to discern the criticality of a newly arrived request. Thus, we identify critical requests using the ML methods. We leverage the feature collector to glean features to facilitate the offline data training. The trained model is applied to elect critical requests at the inference stage. Choosing an appropriate ML model plays a pivotal role in the design of CCache, which enables us to understand the feature of request, predict future trends, and anticipate outcomes. As this model furnishes decision-making procedures in CCache, we ought to analyze the complexity of the ML model, request features, and the algorithm's efficiency.

In this study, we select artificial neural networks (ANNs) [28], deep neural networks (DNNs), K-nearest neighbors (KNNs), random forests (RFs), and decision trees (DTs) from the available neural networks and conventional ML models [29]. The traces used in this paper lack distinct spatial locality or time-series features, excluding complex network structures. In addition, these intricate models may demand more computational resources [30], which are scarce in SSDs. ANNs can perform effectively with ample training data, but these techniques are prone to overfitting amid limited datasets [31]. A KNN model is primarily utilized for non-linear data instead of linear data, as in the case of the traces used in this paper. It is sensitive to outliers, negatively affecting classification accuracy. In multi-classification problems, DTs create complex structures to fit all classes of the training data, which reduces the model's accuracy under a new dataset. In scenarios with unbalanced class distributions, DTs may be biased towards the majority classes at the expense of minority ones. In addition, support vector machines are primarily used for binary classification and require an extension for multi-classification problems. It is challenging to identify the optimal classification hyperplane. The misclassified points can lead to inaccurate classification.

In CCache, we opt for a RF classifier – an integrated learning algorithm based on DTs for predictions. When we perform the classification, new input samples enable each DT in the forest to make judgments and classify. Each DT offers a classification result. A final result is elected by considering the outcome of a DT with the highest number of classifications. When dealing with multiple classification problems, RFs can vote on the results of multiple DTs to improve prediction accuracy. When the number of samples for specific categories is small, the RF can effectively handle such an imbalance to enhance the model's accuracy. Finally, RF is efficient thanks to the capacity of coping with large-scale dataset in a short period: data may be processed in parallel while using only a portion of the features and samples for each DT. The training module in Fig. 4 is divided into n training sample sets, each with m samples selected from the dataset to form a separate sub-dataset. These n sub-models are derived by training the sub-training set. Then, we combine multiple classifiers to obtain a strong classifier.

For the aforementioned reasons, we choose the RF as the ML model for this study. RF has been proven to be an ideal model for multi-classification problems [32]. Furthermore, we evaluate the classification performance of different machine-learning methods, see Table III. The DT has the lowest storage overhead and inference time but lower accuracy than RF. RF provides the highest accuracy, and its online inference overhead is acceptable in CCache. The other models exhibit poor accuracy and inference time.

D. Training and Inference in CCache

1) Training: A training set consists of sample features and labels. The feature collector harvests the features of sampled requests for the training set, but the sampled labels are still missing. First, we collect the feature in the same order as requests are executed. We simulate a mock cache, mirroring

TABLE III
THE CLASSIFICATION RESULTS OF DIFFERENT MODELS

Models	Accuracy	Storage Cost (KB)	Inference Time (μ s)
ANN	0.89	32.4	15.6
DNN	0.75	87.6	38
KNN	0.79	145	29.5
RF	0.93	57.9	11.2
DT	0.82	6.8	3

the size of the actual cache space, offline on the host side. We synthesize the workload using a request stream composed of feature samples. We employ benefits to assist CCache with the cache replacement, as outlined in Eq. 2. Since the request stream in the synthetic workload is transparently public, we derive the following distance of the same LPA. It is the number of requests (Range) between the current and subsequent access. Because recency and frequency are essential factors for temporal locality and hotness, we incorporate these factors in the benefit calculation: the cost-benefit measure of caching each request is gauged. According to this benefit, each page-sized request is prioritized, and we simulate the cache management based on this priority.

$$Benefit = \frac{1}{Recency} * Frequency * \frac{1}{Range} \quad (2)$$

A large value of cost-effectiveness benefit indicates that the request with a high priority can be easily placed in the cache. The small recency indicates that the workload is featured with a good locality. Hence, higher cost-effectiveness is obtained by $1/Recency$. Requests with a high frequency exhibit increased hotness and subsequently yield great cost-effectiveness. The values of recency and frequency are gleaned in the feature collection. The range is the number of I/O requests between the current and subsequent requests that access the same LPA. The lower the number of interval requests, the faster the request is re-accessed. Therefore, requests with a small range are more cost-effective by $1/Range$.

We designed an algorithm based on the offline optimum algorithm (OPT) [33] – Labeler – to obtain labels. When the cache is full in the simulated cache space, we select a request with the lowest benefit to be replaced by a new one. Each request that accesses the cache simulator can be set to be a value – *life*. This metric quantifies the number of accesses between the moment a new request enters the cache and when it is replaced due to having the lowest benefit value. Fig. 5 shows a way of leveraging benefit values to simulate the cache management. New incoming requested data are directly stored in the cache with free space, but a cache replacement occurs when the cache becomes full. The request with the smallest benefit is selected for the first cache replacement (*i.e.*, the request with LPA 2 and benefit 4). Subsequent cache replacements are performed similarly. Meanwhile, we keep track of the *life* value corresponding to each request. We divide the *life* values into critical, intermediate, and non-critical requests. LPAs labeled as non-critical requests will be accessed a few times – only hit once or even not be accessed (*i.e.*, *life*=1). LPAs labeled as critical requests will be accessed frequently in the foreseeable future

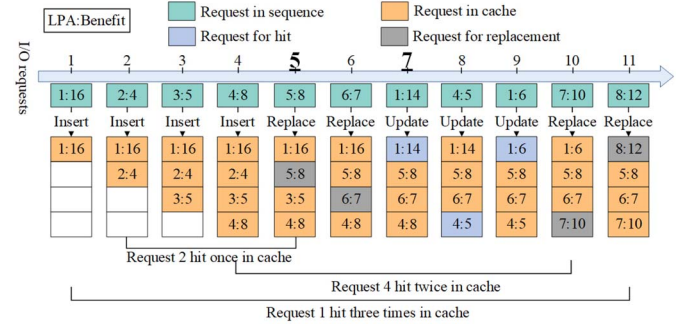


Fig. 5. Using benefit as the cost of cache replacement in a request sequence to obtain label *life* for ML model. At point 5, there is no free space in the cache, and request 2 has the smallest benefits; thus, request 2 is evicted. At point 7, request 1 hits cache, and its benefit value is updated.

(*life* ≥ 3). These intermediate requests – a buffer between the above two-type requests – are introduced to avoid cliffhanging changes between critical and non-critical requests (*i.e.*, *life*=2).

We label different values to the requests according to the ranges of life. The features are combined with the label to assemble the training set, which is fed into the random forest algorithm to construct the ML model. The offline model training does not impact SSD performance. After the offline training is accomplished, the model is delivered to the FTL through the agent module for request processing.

2) *Inference*: The offline models are transferred to the NOR-flash, where the agent stores the firmware inside the SSD. Upon the arrival of a request, the features gathered by the feature collector are fed into the ML model to conduct inference. CCache inputs feature into different decision tree models to obtain unique decision results. Among all the results, the one with the highest proportion is voted as the final label; however, the model stored in the SSD requires the corresponding Python environment and the library for the model calls, which introduces overhead. Random forest-based ML is an integrated learning method consisting of multiple decision trees, where each decision tree is conditionally judged to generate a result. Therefore, we convert the ML model into an “if-else” statement in the C function. Initially, the classification conditions of each decision tree in the random forest are translated to “if-else” conditions, and the result of the judgment is that of the decision tree. Then, we secure all the judgment results and vote to yield a final label. In the process, the features are employed as parameter inputs to the function. We use the method to abstract the model into a C function, and label values are acquired through the function’s return. The label values are utilized for cache management.

E. Cache Management

We employed read-write caching schemes that can store both read and write requests in this paper. First, the complexity of I/O access patterns poses challenges in achieving optimal performance across a wide range of workloads. Thus, a simple read or write caching scheme struggles with mixed read-write workloads. Second, the I/Os access patterns may change throughout

a workload, and a read-intensive workload may present write-intensive I/Os within a short time window. Hence, when dealing with read- and write-intensive workloads, a simple read or write caching scheme may potentially degrade performance. Thus, the read-write caching scheme can achieve better adaptability and higher performance. Several prior studies have employed read-write caching schemes [8], [9], [10], [34] to balance the trade-offs between read and write caching to reshape overall system performance.

Request Filtering. Fig. 4 plots that labels are procured corresponding to the three types of requests using the trained machine-learning model. We perform initial filtering when non-critical requests enter the SSD.

Given a non-critical request, CCache determines whether to store it in the cache. When the request is hit in the cache, it is directly served by the cache. Otherwise, if the non-critical request is a read, data is retrieved from NAND flash. The write request can be written into flash memory or cache, and the former method fully utilizes the cache space with an improved hit rate. However, not all requests labeled as non-critical written into the flash memory. This is because numerous I/Os – such as GC and erases – conflict within flash memory, consume resources, and contend with write requests. Therefore, we design a chip-level monitoring unit to detect these requests. This unit monitors the status of each chip – busy or idle. If this unit recognizes, while processing a new request, that the chip to be written back is busy, CCache writes the new request to the cache; otherwise, the request is written back to flash memory. This approach prevents I/O requests from colliding with underlying flash memory operations, thereby eliminating waiting time. We place critical and intermediate requests in the cache.

Request Management. After a request is handled by the request filtering module, the request in the cache are managed according to their label values. In CCache, we design tri-level linked lists – linked lists L1, L2, and L3 to organize different types of requests. When the label value of a request is non-critical, intermediate, and critical, we put the request into the L1, L2, and L3, respectively. We advocate for the tri-level linked list because when the cache replacement occurs, requests are evicted according to the request priority. More specifically, a tail request in L1 is evicted first. In case of an empty L1, we expel tail requests in L2. If both L1 and L2 are vacant, requests in L3 are forced out. When the request hits the cache, the label value of a request may have changed. In such cases, the hit cache node is moved to the head of the corresponding linked list based on its current label value. For instance, if a request in L1 with a critical label value is hit, but its label value has changed to the intermediate, the hit cache node is moved to the head of L2. Similar adjustments are made for the other cases. In the event of a cache miss, the request is inserted into the head of the corresponding linked list based on the label. For example, if the request label is critical, it ought to be inserted into the head of L1. As the label of a request changes, the data in each linked list are transferred to another linked list to reflect their updated labels.

Algorithm 1 CCache-based cache replacement algorithm

Require: page requests $LPA1, LPA2, \dots, LPAi \dots$

```

1: procedure CCache-BASED SCHEMES
2:   if  $LPA$  hits node  $i$  in the cache then
3:     Update_label_node( $i$ ) /* The label of the node changes and needs
       to be updated */
4:     Update_linked_list( $L1, L2, L3$ ) /* Insert the request into the cor-
       responding linked list according to the label */
5:     if page request is write one then
6:       Update_request_node( $i$ ) /* Once it is a write request, the data
       changes and needs to be updated */
7:     end if
8:   else
9:     if Cache is full then
10:      if  $L1$  is not empty then
11:        Evict_tail_request( $L1$ ) /* Once the cache is full, we evict
       the node based on the priority */
12:      else
13:        if  $L2$  is not empty then
14:          Evict_tail_request( $L2$ )
15:        else
16:          Evict_tail_request( $L3$ )
17:        end if
18:      end if
19:      Write_to_cache( $LPAi$ )
20:      Update_linked_list( $L1, L2, L3$ )
21:    else
22:      Write_to_cache( $LPAi$ )
23:      Update_linked_list( $L1, L2, L3$ )
24:    end if
25:  end if
26: end procedure

```

The three linked lists offer a good distinction between hot and cold requests. In contrast to a regular linked list, the tri-level linked list offers enhanced retrieval capacity with precise replacement of the least frequently accessed requests in the cache. The processed request only enters the cache management module, which processes the request according to its label value. In Algorithm 1, Lines 2-7 represent the situation where LPA hits the cache, we need to update the node's label and linked list. For write requests, data information must also be updated. Lines 9-20 present the procedure for handling a cache miss when the cache is full. Initially, we elect a cached request to be replaced based on its priority. Subsequently, the current request is inserted into the cache, and the linked list is updated accordingly. Lines 22-23 outline the steps taken when the requested LPA is not located in the cache that has available space. In this scenario, the request is written to the cache followed by an adjusted linked list.

The eviction policy of CCache has strictly followed the priority order of L1, L2, and L3. The eviction priority of L3 is low; thus, there is a small likelihood for a request residing in L3 to be evicted when cache replacement occurs. Since CCache's ML inference results might be inaccurate, some erroneous requests exist in L3 – the cache space is occupied for a long period. Then, it takes a long time to evict these requests from the cache. To address this issue, we design an active degradation strategy. CCache monitors the request at the end of L3. If these requests are not accessed for an extended period, they are downgraded. They are put into the head of L2, and for the data in L2, we take the same action and put it into the head of L1.

TABLE IV
CONFIGURATION PARAMETERS OF SSDSIM

SSD Organization	Capacity: 32 GB, four channels, two chips per channel
Flash Micro-architecture	8 KB per page, 64 pages per block, 2048 blocks per plane, two planes per die, two dies per chip
Flash latency	Read: 20 μ s, Program: 200 μ s, Erase: 1.5ms.
FTL	GC Threshold: 0.3, Mapping scheme: page-level mapping, aged:0.5

V. EXPERIMENTAL SETUP AND EVALUATION

A. Experimental Setup

Platform. We leverage a popular SSD simulator — SSDsim [35] — to implement CCACHE. Table IV lists the experimental configurations. We set the parameter aged to 0.5 to represent the percentage of invalid pages when the test runs. We conduct experiments using seven cache sizes under real-world enterprise traces, including three traces in MSR and five traces in the Systor’17 Traces collected from the Fujitsu Labs in Japan. The eight traces exhibit a diversity of temporal locality and critical request rates. CCACHE trains an ML model in the ML library Scikit-learn [36]. The library includes an assorted of practical tools for data analytics. We initiate a decision-tree structure for the trained model in the inference stage. We implement a function using the C programming language for each decision tree. Then, these C functions are invoked by a random forest function before voting is administered. We derive labels by inputting features into the random-forest function.

Comparison Cache Schemes. We evaluate the performance of LRU, CFLRU [8], LCR [9], NCACHE [13], ML_WP [22] CCACHE_A, and OPT [33] cache schemes. NCACHE is a machine-learning-based cache scheme to improve the cache hit rate by predicting valid and invalid requests. ML_WP is an ML based write policy that reduces writes to SSDs by recognizing write-only data. CCACHE_A replaces the training model of CCACHE with ANN to validate the efficacy of CCACHE. OPT, a theoretical optimum cache scheme, selects request pages for cache replacement based on the expectation that the pages will not be accessed for the longest duration. OPT ensures that pages stored in the cache are highly likely to be reused in the future, maximizing the cache hit rate.

Performance Metrics. **Average response time** represents a time interval between an I/O request entering the SQ queue and the moment when a host receives feedback from the SSD. We normalize the average response time and erase count to investigate the performance of the caching schemes, and the LRU-based results are assessed as a normalized benchmark. **Write amplification (WA)** is a ratio between the amount of written data to flash memory and the request volume issued by a host. WA captures the influence of the cache schemes on SSD lifetime. **Erase count** in the flash memory quantifies the SSD lifetime because each block in the flash memory has a finite maximum number of erase operations. **Write/read hit rate**



Fig. 6. The y-axis represents Avg. RT, normalized to that of the LRU under traces. The x-axis denotes cache sizes.

(**WHR/RHR**) is the rate at which an I/O request hits the cache, and a high hit rate reduces the response time.

B. Evaluation

1) **Average Response Time (avg. RT):** Fig. 6 shows that CCACHE has a clear edge over the other cache schemes under traces. For a 16-MB cache, CCACHE reduces avg. RT of LRU, CFLRU, LCR, NCACHE, ML_WP, and CCACHE_A by averages of 19.1%, 21.2%, 15.4%, 16.6%, 19.6%, and 14%, respectively. When it comes to the 32-MB cache, CCACHE delivers an average improvement of 19%, 23.9%, 14.8%, 17.5%, 19.6%, and 13.4% over the same schemes, respectively. Under LUN2-4 and an 8-MB cache, CCACHE curbs avg. RT by averages of 40.1%, 39.7%, 35.6%, 34.7%, 39.9%, and 36% compared with LRU, CFLRU, LCR, NCACHE, ML_WP, and ANN, respectively. The overall temporal localization of LUN2-4 and the critical request rate are both significantly high. Nevertheless, there is a substantial standard deviation in the phase. As a result, the phase localization and critical requests exhibit an immense fluctuation, which leads to the loss of critical requests. CCACHE revamps performance by retaining critical requests and evicting non-critical ones. Compared against NCACHE, CCACHE slashes the avg. RT by up to 35.2% with averages of 8.93%, 14.3%, 14.1%, 16.3%, 16.6%, 17.7%, and 17.5% in the 1-, 2-B, 4-, 8-, 16-, 32-, and 64-MB cache, respectively. Although NCACHE utilizes the differentiation between valid and invalid request eviction to enhance the performance, NCACHE is unable to recognize critical requests, which eventually gives rise to performance bottlenecks. Compared with CCACHE_A, CCACHE curtails the avg. RT by up to 36.4% with averages of 4.9%, 11.5%, 12.8%, 14.6%, 14%, 15.3%, and 13.4% for cache sizes ranging from 1 MB to 64 MB, respectively.

The ineffectiveness of the ANN model diminishes its accuracy to obtain critical requests, leading to high avg. RT. In stg_0, CCACHE with 1-MB cache share the similar performance with that of OPT: the discrepancy is 3.1%. Given 8-MB cache for



Fig. 7. The y-axis represents WA, normalized to that of LRU.

LUN2-4, the avg. RT of LRU, CFLRU, LCR, NCache, ML_WP, and CCACHE_A is longer than OPT by 44.2%, 43.9%, 40.9%, 39.1%, 44%, and 40.3%, respectively; however, the difference in avg. RT between CCACHE and OPT is merely 6.9%, making CCACHE a clear champion among its peers.

2) *SSD Lifetime*: We investigate the impact of the caching schemes on SSD lifetime through WA and erase count. Fig. 7 reveals WA under the various traces and cache sizes. CCACHE reduces WA by up to 58% and 46% under LUN2-1 and LUN2-4, respectively. Under LUN0-1, 32- and 64-MB CCACHE have similar WAs that are close to OPT. Given the 2-MB cache, CCACHE trims WAs of LRU, CFLRU, LCR, NCache, ML_WP, and CCACHE_A by averages of 24.7%, 30.2%, 26.1%, 22.2%, 26.2%, and 16.3%, respectively. In the case of the 32-MB cache, CCACHE reduces WA by 11%, 11%, 14.3%, 8.6%, 11.6%, and 8%. Under traces LUN2-2 and LUN2-4, the critical request rate varies during the phase, where CCACHE shows optimized performance. Under LUN2-3, the small-sized cache (2 MB)-enabled CCACHE lowers WA by averages of 36.2%, 50.2%, 41.8%, 30.9%, 39.8%, and 30.8% compared with LRU, CFLRU, LCR, NCache, ML_WP, and CCACHE_A, respectively. Under LUN2-4 and 2-MB cache, CCACHE cuts back the WA by 29.8%, 46.6%, 30%, 24.9%, 35.8%, and 22.6% than the same schemes, respectively. In addition, an increasing cache size curtails the WA.

Fig. 8 unravels that given the 2-MB cache, CCACHE reduces erase count by averages of 42.5%, 51.5%, 45.6%, 38%, 39.6%, and 29.6% compared against LRU, CFLRU, LCR, NCache, ML_WP, and CCACHE_A, respectively. With the 4-MB cache, CCACHE scales down erase count of LRU, CFLRU, LCR, NCache, ML_WP, and CCACHE_A by averages of 25.1%, 31.3%, 30.9%, 22.7%, 25.8%, and 12.9%, respectively. Under LUN2-2 and LUN2-4, CCACHE slashes erase count by up to 66.1% and 63.5%. Under LUN2-1, the 4- and 16-MB CCACHES' erase count is close to that of OPT. CCACHE protects critical requests and reduces the frequent write-back of critical requests back to flash memory, thereby lowering erase count for SSDs. Under ts_0, we observe that erase count remains at 0 when the cache size is large, whereas the erase count stays low because

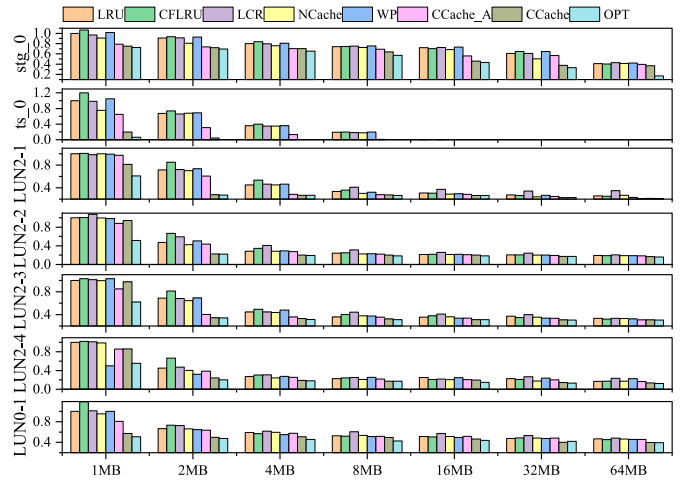


Fig. 8. The y-axis represents erase count, normalized to that of LRU.

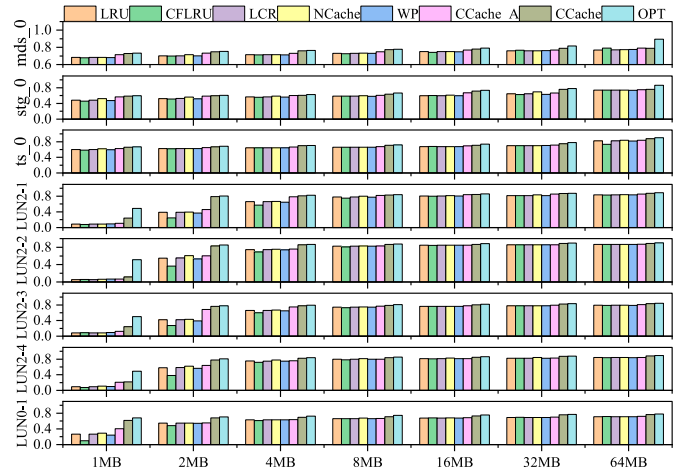


Fig. 9. The y-axis denotes WHR of caching schemes under traces.

the small-sized workload has sound localization. CCACHE has a clear edge over LRU, CFLRU, LCR, NCache, ML_WP, and CCACHE_A because erase count is small.

3) *Hit Ratio*: Fig. 9 plots the WHR with 1-MB cache, CCACHE optimizes the hit ratios of LRU, CFLRU, LCR, NCache, ML_WP and CCACHE_A by averages of 41.1%, 46.5%, 40.9%, 37.4%, 39.4%, and 24.7%. Given the 4-MB cache, the hit-ratio improvement offered by CCACHE becomes 19.6%, 14.9%, 19.6%, 9.4%, 11.2%, and 5.5%. Under the LUN0-1 workload, CCACHE pushes the maximum write hit to an impressive level of 84%. The primary objective of CCACHE is to safeguard critical requests from being evicted from the cache. When the number of reads to the critical data surges, the hit rate hikes. The figure shows that the WH of CCACHE increases with the cache size, and both measures are close to those of the OPT.

Fig. 10 reveals RHR of tested cache schemes. In the 8-MB cache, CCACHE outperforms LRU, CFLRU, LCR, NCache, ML_WP, and CCACHE_A by averages of 11.3%, 22.2%, 21.2%,

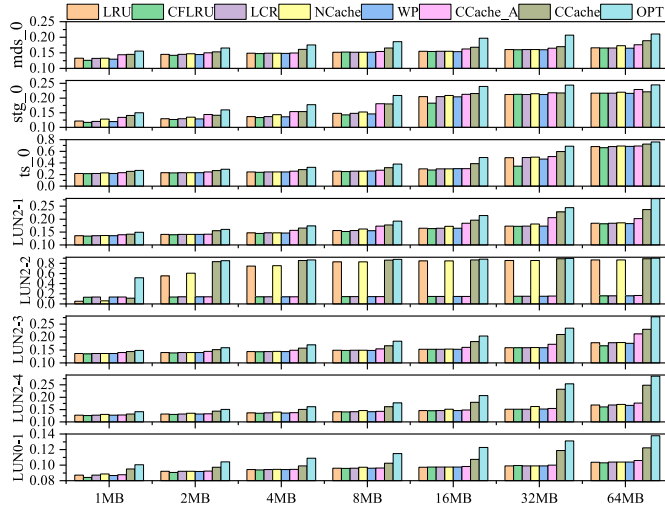


Fig. 10. The y-axis denotes *RHR* of caching schemes under traces.

10%, 21.4%, and 16.5%, respectively. Regarding the 32-MB cache, CCACHE optimizes LRU, CFLRU, LCR, NCache, ML_WP, and CCACHE_A by 16.2%, 29.2%, 26%, 14.8%, 26.6%, and 22%, respectively. Under the LUN2-2 condition, CCACHE's read hit ratio is higher than those of CFLRU, LCR, and CCACHE_A by up to 83.6%, 83.4%, and 83.3%, respectively. CCACHE's hit-ratio improvement is expected because CFLRU and LCR are conditional cache schemes: CFLRU evicts clean requests whereas LCR replaces pages in a low-load chip – and these pages are frequently read. The cache read hit rate of CFLRU and LCR is deteriorating when replacing these requests from the cache. CCACHE and ANN share the same cache structure, but the difference lies in the classification of requests. CCACHE_A replaces the training model of CCACHE with ANN. Due to low classification accuracy, CCACHE_A obtains the wrong request types. When cache replacement occurs, CCACHE_A can evict the correct critical requests from the cache early, decreasing the hit rate.

C. Extended Studies

1) *Critical Request Rate*: The critical request rate is a ratio between the number of critical requests to the total ones in the cache. This metric quantifies the cache utilization in a cache scheme. Notably, a large critical request rate signifies high cache utilization. Fig. 11 depicts the critical request rates of LRU, CFLRU, LCR, NCache, and CCACHE managing the 32-MB cache. The x-axis and y-axis represent the time slots and the critical request rate. The critical request rate of CCACHE is higher than those of the other cache schemes: CCACHE is a front runner in terms of cache utilization among its peers. CCACHE is undoubtedly a winner because when more critical requests exist in the cache, more requests are fulfilled by the cache, thereby enhancing the cache hit rate and curbing the number of writes to the flash memory. The critical request rate of NCache is higher than those of LRU, CFLRU, and LCR because NCache prevents valid requests from being evicted from the cache. A portion of valid requests exist in the critical requests. The critical request

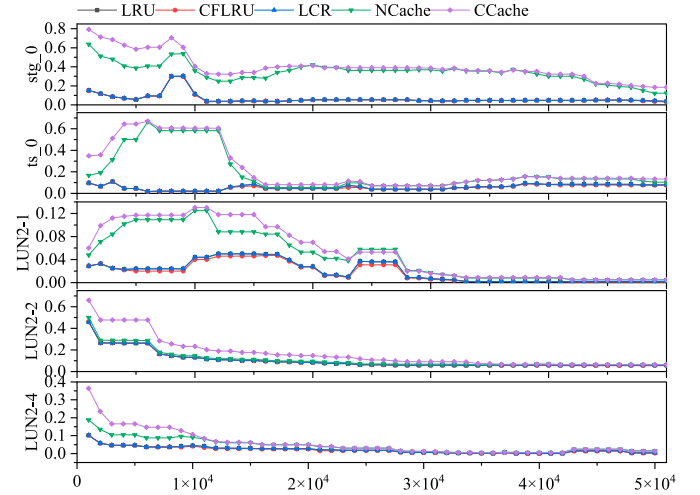


Fig. 11. Critical request ratio in caching schemes. X-axis represents request sequence in the workload, and the y-axis denotes the critical request ratio.

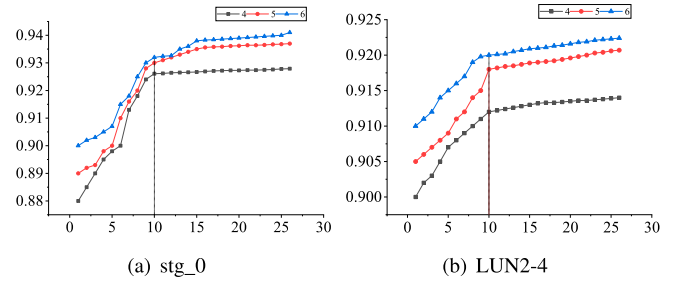


Fig. 12. Accuracy of the model at the depth of 4, 5, and 6 under stg_0 and LUN2-4. The x- and y-axis denote numbers of trees and accuracy, respectively.

rate of CCACHE goes up when there is a high degree of localization because CCACHE preferably evicts non-critical requests. Facing a low localization, the number of critical requests of the workload becomes limited; therefore, the possibility of evicting critical requests is slim across all the cache schemes. Thus, there is a negligible disparity in the critical rate among the tested cache schemes.

2) *Model Parameters*: Generally, increasing the depth of the random forest model fits data more closely at the cost of computational complexity, reduced speed, and the overfitting problem [37]. There are certain constraints on the number of trees that can be used in a random forest model. As the number of trees increases, it converges to a lower generalization error. When the model's accuracy has reached convergence, further increasing the number of trees doesn't improve accuracy; rather, rising computational overhead is inevitable. Fig. 12 illustrates the relationship between the accuracy and the number of trees in the context of different tree depths. The model accuracy is optimized with an increasing number of trees. When the number of trees is 10, the accuracy converges. Afterward, there is not too much change. Meanwhile, we study the accuracy of different tree depths, and the accuracy of the tree depth of 5 is significantly improved compared with that of the tree depth of

TABLE V
THESE MIXED TRACES USED IN THE EXPERIMENT

MIX1	LUN2-1 LUN0-1	MIX2	LUN2-1 LUN2-2
MIX3	LUN2-3 LUN0-1	MIX4	LUN2-4 LUN0-1
MIX5	stg_0 ts_0		

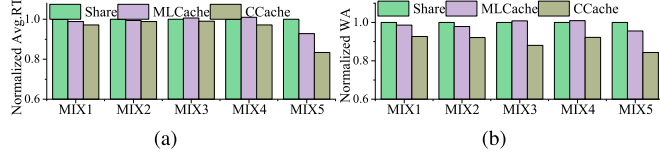


Fig. 13. Performance of cache schemes under mixed workloads.

4. In contrast, the six-tree-depth accuracy is enhanced compared to the tree depth of 5. Thus, we set the tree depth to 5 in CCache, and the number of trees is referred to as 10, which enables the stochastic model to achieve a high accuracy while mitigating the risk of overfitting.

3) *Performance of CCache Under Mixed Workloads:* ML-Cache employs a neural network model to learn the impact of reuse distance on cache allocation across multiple traces. It also introduces a parallel write-back scheme to enhance workload fairness. We integrated MLCache in SSDSim to simulate multi-stream workloads by synthesizing existing traces, see Table V. We compared the performance of CCache with MLCache and a baseline Share scheme, where all streams share the entire cache space. In Fig. 13(a), CCache reduces *avg. RT* by 5% and 3% over Share and MLCache, respectively. Under MIX5, CCache achieves a maximum improvement in *avg. RT* of 16.6% and 12.2% over Share and MLCache, respectively. Fig. 13(b) indicates that, on average, CCache achieves a 10.1% and 9% reduction in WA over Share and MLCache, respectively. The maximum reduction in WA by CCache is 15.6% over the Share strategy under MIX5 and 12.7% over MLCache under MIX3.

4) *Accuracy of Different Labels:* We gauge the accuracy of ANN, KNN, and RFC models. Table VI demonstrates that the accuracy of RFC is 4.4% and 22% higher than ANN and KNN on prediction for non-critical requests. For these intermediate requests, RFC outperforms ANN and KNN by 4.3% and 29.6%. RFC improves critical requests in ANN and KNN by 2% and 7.3%, respectively. RFC's accuracy is 4% and 14.89% higher than those of ANN and KNN. False prediction leads to inaccurate request determination. For the response time, CCache presents a response time that is shorter than CCache_A. The accuracy of KNN is too low to warrant any usage in CCache.

5) *Spatial Locality:* As mentioned in Section IV-B1, we studied the temporal locality instead of the spatial characteristics in workloads in CCache. In this section, we evaluated the impact of disregarding spatial locality on garbage collection (GC) copy operations. In Fig. 14, the empirical study results, depicted with the caching schemes on the x-axis and total SSD writes on the y-axis, reveal that CCache outperforms LRU, CFLRU, LCR, NCache, ML-WP, and CCache_A in terms of

TABLE VI
COMPARISON OF DIFFERENT LABELS USING ML MODELS

Model	Accuracy			
	Non-critical requests	Intermediate requests	Critical requests	Total requests
RFC	0.925	0.912	0.932	0.93
ANN	0.884	0.872	0.913	0.893
KNN	0.721	0.642	0.864	0.792

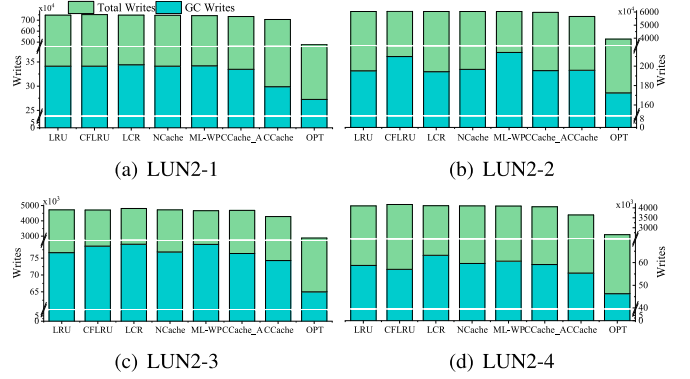


Fig. 14. Total writes to flash memory and writes due to GC.

TABLE VII
ACCURACY OF CCache WITH VARIOUS FEATURE COMBINATIONS

	CASE1	CASE2	CASE3
Feature Combination	recency, frequency, period, size	recency, frequency, period, distance	recency, frequency, size, distance
Accuracy	0.812	0.844	0.773
	CASE4	CASE5	CASE6
Feature Combination	recency, period, size, distance	frequency, period, size, distance	recency, size, frequency, period, distance
Accuracy	0.796	0.828	0.93

write reduction, achieving an average decrease of 7.8%. Specifically, CCache reduces GC-induced writes by 5.2%, 6.9%, 7.7%, 5.8%, 8.9%, and 4.9%, compared with LRU, CFLRU, LCR, NCache, ML-WP, and CCache_A, respectively, with a maximum reduction of 13.2%. Under the LUN2-1 workload, CCache's performance closely aligns with the OPT (optimal) scheme compared with the other schemes. This improvement is attributed to CCache's exploitation of temporal locality that helps in reducing extra writes.

6) *The Significance of the Five Features in CCache:* In our test, we explored the impact of various feature combinations on accuracy, see Table VII. The configurations, namely CASE1, CASE2, CASE3, CASE4, and CASE5, encompass various subsets of features, while CASE6 includes all features. The comparative analysis reveals that the accuracy discrepancy between other feature combinations and the comprehensive feature set (CASE6) ranges from a minimum of 0.086 to a maximum of 0.157, with an average deviation of 0.12. This indicates the significance of each feature in achieving optimal

accuracy. Consequently, we incorporated all the features in our study to achieve the results with high accuracy.

VI. RELATED WORK

Cache plays a crucial role in improving SSD performance, and prior studies have proposed cache schemes to organize and manage requests in the cache. In this study, we classified cache schemes into traditional algorithms and ML-based algorithms. The traditional cache schemes include the front-end, back-end, and combined two-end schemes.

A. Traditional Cache Schemes

Front-end. Front-end cache schemes primarily focus on I/O access patterns in workloads. These cache schemes are designed according to user I/O access features — hot/cold, temporal locality, and I/O intensity. LRU, a widely popular algorithm, utilizes the temporal locality. LRU places the most recently accessed requests at the head of a linked list, whereas the longest unaccessed requests are situated at the tail of the linked list. However, this method is susceptible to periodic I/O access patterns and relies on the temporal locality. LFU [38], which replaces cache requests according to I/O access frequency in a period, prioritizes the least frequently accessed requests for eviction. However, LFU is prone to evicting cache requests that have recently entered the cache. LFU, ignoring the timeliness of requests, may cache old requests. In VS-Batch [39], temporal and spatial locality are characterized through the visibility graph method. It adaptively generates a viewable figure of cache requests. VS-Batch can adjust neighboring or nearby cache request pages according to the connections in the viewable figure.

Back-end. Researchers designed cache schemes anchored on the features of flash memory. CFLRU [8] exploits the read/write imbalance inside flash memory by prioritizing clean requests during cache replacement, reducing the number of writes to flash memory. Unfortunately, these cache schemes ignore the fact that there are few clean requests under write-intensive workloads, deteriorating the cache hit rate. Therefore, researchers generally set a window to select clean requests. For example, LCR [9] and ECR [34] introduce the busy state in a flash chip queue and address the access conflict between user I/Os and flash I/Os. When cache replacement occurs, requests in the low-load chip are prioritized to be replaced, which reduces response time. Garbage collection includes read, write, and erase operations, leading to conflicts with external I/Os. GCaR [10] proactively detects these conflicts and prioritizes request evictions that do not conflict with the GC I/Os during cache replacement, shortening response time. These cache schemes, however, decrease the cache hit rate due to the constant filtering out of potentially suitable requests.

Combining front-end and back-end. Some prior work incorporated I/O access patterns of workloads (spatiotemporal locality) and the features of flash memory inside SSDs. For instance, CCF-LRU [40] adopts the hot-cold states of I/O requests coupled with the hit cost of cache. It prioritizes cold-clean

requests amid eviction, followed by hot-clean requests. Co-Active [12] – a collaborative and proactive write-back scheme – collaborates I/O access patterns with the state of flash chips. When the arrival times of I/O requests are widely separated, dirty requests are flushed to the flash memory, achieving an sufficient number of clean requests in the cache. In the presence of burst workloads, clean requests can be deleted to reduce response time. DAC [11], which divides the cache area into hot and cold regions, dynamically adjusts the ratio of the two areas according to changing I/O access patterns. During cache replacement, clean requests in the cold region are prioritized. However, all these cache schemes require specific parameter tuning: these parameters ought to be obtained through statistical methods – being unable to sense future I/O trends.

B. Machine-Learning-Based Cache Schemes

A decision tree-based classifier is designed using the “one-time access” criterion. The relevant information is used as features, and cost-sensitive learning is applied to improve classification accuracy, which reduces write operations and improves the SSD lifetime. NCache advocates for a decision tree model to identify valid and invalid requests. By prioritizing replacing invalid requests within a window, minimizing the number of writes to flash memory. A pipelined-mode parallel approach is devised to suppress the overhead of ML techniques. Chandranil et al. [41] proposed a prefetching approach to tackle the performance bottlenecks: this approach predicts LBAs in advance to curb the read latency in flash memory and the inference latency of the model. ML-CLOCK [42] enhances sequential-write performance in flash-based storages with a single-level perceptron (SLP) for online learning in cache eviction. While low in overhead, its accuracy may decrease in multiple classification scenarios. SLP is best suited for binary classification, but may not be ideal for multiple classifications required for cache eviction decision-making.

The online reinforcement learning (RL) method offers adaptability over offline training methods in cache replacement decision-making. Online RL can adjust cache policy (*e.g.*, CACHEUS [43], Cache-MAB [44]) to adapt well to complex and dynamic I/Os access patterns compared with the prepared training in advance. For example, CACHEUS utilizes a combination of experts designed to address workload primitive types. While CACHEUS adopts an online RL based on regret minimization, this approach presents challenges in the context of SSDs. CACHEUS leverages the RL to dynamically select the optimal cache replacement policy, which demands real-time updating and tuning decisions based on historical results. This process incurs extra computational overhead and consumes limited resources inside SSDs. Cache-MAB’s effectiveness hinges on the availability of workload content and context information, which may not be accessible in SSDs. Additionally, Cache-MAB is designed for multi-level caches in routers, whereas SSDs feature a single-level cache. A multi-level cache in SSDs would incur extra costs and may not align with the primary purpose of the cache inside SSDs. It is not suitable to apply Cache-MAB in SSDs.

The trial-and-error approach in the RL may degrade performance in resource-constrained SSDs, and it is challenging to apply online RL in SSDs. First, an online RL-based cache scheme requires interaction between the host and the device. While some advanced schemes [43] use coarser granularity to reduce overhead, this compromises accuracy. Second, this online RL-based cache scheme can incur losses during the initial learning phase before an effective policy is established.

RFs based on training data make decisions by constructing multiple DTs. RFs can quickly achieve results based on input features. The real-time information collected by CCache can be easily obtained from realistic traces. Meanwhile, it can also be swiftly calculated in a complex environment. In CCache, we design an active degradation policy to evict the data that is correctly predicted to enter the cache, which avoids the loss of accuracy due to changes in I/Os access patterns. In previous work [13], [14], [22], there are some traditional ML models-based caching schemes.

VII. CONCLUSION

We proposed a cache management technique – CCache – driven by a machine-learning model – random forest classifier. CCache is slated to collect features of I/O requests to be applied as inputs to the ML model that classifies I/O requests into critical, intermediate, and non-critical camps. CCache prevents critical requests from being evicted while actively working to force out non-critical requests. More prosaically, the CCache method inhibits non-critical requests from displacing critical requests in the cache. Consequently, the cache serves critical requests with an improved hit rate. We validate CCache through the extensive experiments. For example, under the LUN2-4 trace, CCache slashes the response time by up to 42% with an average of 25.1%. CCache is adroit at curtailing the erase count by an average of 25% with a maximum of 63.5%. Moreover, CCache bolsters the write amplification performance over the existing schemes by up to 46.0%. The performance gap between CCache and OPT, as measured in terms of response time, erase count, and write amplification is merely 13.0%, 14.3%, and 7.0%, respectively.

REFERENCES

- [1] Z. Yang et al., "Autotiering: Automatic data placement manager in multi-tier all-flash datacenter," in *Proc. 36th Int. Perform. Comput. Commun. Conf. (IPCCC2017)*, 2017, pp. 1–8.
- [2] B. S. Kim, J. Choi, and S. L. Min, "Design tradeoffs for SSD reliability," in *Proc. 17th USENIX Conf. File Storage Technol. (FAST 19)*, 2019, pp. 281–294.
- [3] M. Tarihi, H. Asadi, A. Haghdoust, M. Arjomand, and H. Sarbazi-Azad, "A hybrid non-volatile cache design for solid-state drives using comprehensive I/O characterization," *IEEE Trans. Comput.*, vol. 65, no. 6, pp. 1678–1691, Jun. 2016.
- [4] M. Xie et al., "Checkpoint aware hybrid cache architecture for NV processor in energy harvesting powered systems," in *Proc. 11th IEEE/ACM/IFIP Int. Conf. Hardware/Softw. Codesign Syst. Synthesis*, 2016, pp. 1–10.
- [5] H. Lin et al., "Adaptive management with request granularity for DRAM cache inside nand-based SSDs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 42, no. 8, pp. 2475–2487, Aug. 2023.
- [6] L. Shi, K. Qiu, M. Zhao, and C. J. Xue, "Error model guided joint performance and endurance optimization for flash memory," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 33, no. 3, pp. 343–355, Mar. 2014.
- [7] 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.
- [8] 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. Synthesis Embedded Syst.*, 2006, pp. 234–241.
- [9] C. Liu et al., "LCR: Load-aware cache replacement algorithm for flash-based SSDs," in *Proc. IEEE Int. Conf. Netw., Archit. Storage (NAS)*, 2018, pp. 1–10.
- [10] 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. Supercomputing*, 2016, pp. 1–12.
- [11] H. Sun, S. Dai, J. Huang, Y. Yue, and X. Qin, "DAC: A dynamic active and collaborative cache management scheme for solid state disks," *J. Syst. Archit.*, vol. 140, 2023, Art. no. 102896.
- [12] H. Sun, S. Dai, J. Huang, and X. Qin, "Co-active: a workload-aware collaborative cache management scheme for NVMe SSDs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 6, pp. 1437–1451, Jun. 2021.
- [13] H. Sun, Q. Cui, J. Huang, and X. Qin, "NCache: A machine-learning cache management scheme for computational SSDs," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 42, no. 6, pp. 1810–1823, Jun. 2023.
- [14] H. Wang, X. Yi, P. Huang, B. Cheng, and K. Zhou, "Efficient SSD caching by avoiding unnecessary writes using machine learning," in *Proc. 47th Int. Conf. Parallel Process.*, 2018, pp. 1–10.
- [15] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, "Temporal locality in today's content caching: Why it matters and how to model it," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 5, pp. 5–12, Nov. 2013.
- [16] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proc. 14th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2009, pp. 229–240.
- [17] J. Hu, M. Xie, C. Pan, C. J. Xue, Q. Zhuge, and E. H.-M. Sha, "Low overhead software wear leveling for hybrid PCM + DRAM main memory on embedded systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 4, pp. 654–663, Apr. 2015.
- [18] L. Shi, K. Wu, M. Zhao, C. J. Xue, D. Liu, and E. H.-M. Sha, "Retention trimming for lifetime improvement of flash memory storage systems," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 35, no. 1, pp. 58–71, Jan. 2016.
- [19] Q. Li, L. Shi, C. J. Xue, Q. Zhuge, and E. H.-M. Sha, "Improving LDPC performance via asymmetric sensing level placement on flash memory," in *Proc. 22nd Asia South Pacific Des. Automat. Conf. (ASP-DAC)*, 2017, pp. 560–565.
- [20] R. Liu, D. Liu, X. Chen, Y. Tan, R. Zhang, and L. Liang, "Self-adapting channel allocation for multiple tenants sharing SSD devices," *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, vol. 41, no. 2, pp. 294–305, Feb. 2022.
- [21] W. Liu, J. Cui, J. Liu, and L. T. Yang, "MLCache: A space-efficient cache scheme based on reuse distance and machine learning for NVMe SSDs," in *Proc. 39th Int. Conf. Comput. Aided Design*, 2020, pp. 1–9.
- [22] Y. Zhang et al., "A machine learning based write policy for SSD cache in cloud block storage," in *Proc. Des., Automat. & Test Europe Conf. & Exhib. (DATE)*, 2020, pp. 1279–1282.
- [23] J. Miles, "R-squared, adjusted r-squared. Encyclopedia of statistics in behavioral science., 2005.
- [24] "MSR Cambridge Traces," 2008. Available: <http://iotta.snia.org/traces/388>
- [25] C. Lee, T. Kumano, T. Matsuki, H. Endo, N. Fukumoto, and M. Sugawara, "Understanding storage traffic characteristics on enterprise virtual desktop infrastructure," in *Proc. 10th ACM Int. Syst. Storage Conf.*, 2017, pp. 1–11.
- [26] 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.
- [27] H. Kim, M. Ryu, and U. Ramachandran, "What is a good buffer cache replacement scheme for mobile flash storage?" *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 40, no. 1, pp. 235–246, 2012.
- [28] S. S. Miriyala, V. R. Subramanian, and K. Mitra, "TRANSFORM-ANN for online optimization of complex industrial processes: Casting process as case study," *Eur. J. Oper. Res.*, vol. 264, no. 1, pp. 294–309, 2018.
- [29] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*. Cambridge, MA, USA: MIT Press, 2012.

- [30] M. J. Kearns, *The Computational Complexity of Machine Learning*. Cambridge, MA, USA: MIT Press, 1990.
- [31] S. Agatonovic-Kustrin and R. Beresford, "Basic concepts of artificial neural network (ANN) modeling and its application in pharmaceutical research," *J. Pharmaceutical Biomed. Anal.*, vol. 22, no. 5, pp. 717–727, 2000.
- [32] M. Fernández-Delgado, E. Cernadas, S. Barro, and D. Amorim, "Do we need hundreds of classifiers to solve real world classification problems?" *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 3133–3181, 2014.
- [33] L. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Syst. J.*, vol. 5, no. 2, pp. 78–101, 1966.
- [34] H. Chen, Y. Pan, C. Li, and Y. Xu, "ECR: Eviction-cost-aware cache management policy for page-level flash-based SSDs," *Concurrency Comput. Pract. Exp.*, vol. 33, no. 15, 2021, Art. no. e5395.
- [35] Y. Hu, H. Jiang, D. Feng, L. Tian, H. Luo, and S. Zhang, "Performance impact and interplay of SSD parallelism through advanced commands, allocation strategy and data granularity," in *Proc. Int. Conf. Supercomputing*, 2011, pp. 96–107.
- [36] F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, no. null, pp. 2825–2830, 2011.
- [37] S. Safavian and D. Landgrebe, "A survey of decision tree classifier methodology," *IEEE Trans. Syst., Man, Cybern.*, vol. 21, no. 3, pp. 660–674, May/Jun. 1991.
- [38] J. T. Robinson and M. V. Devarakonda, "Data cache management using frequency-based replacement," *SIGMETRICS Perform. Eval. Rev.*, vol. 18, no. 1, pp. 134–142, Apr. 1990.
- [39] Z. Sha, Z. Cai, F. Trahay, J. Liao, and D. Yin, "Unifying temporal and spatial locality for cache management inside SSDs," in *Proc. Proc. Des., Automat. & Test Europe Conf. & Exhib. (DATE)*, 2022, pp. 891–896.
- [40] Z. Li, P. Jin, X. Su, K. Cui, and L. Yue, "CCF-LRU: A new buffer replacement algorithm for flash memory," *IEEE Trans. Consum. Electron.*, vol. 55, no. 3, pp. 1351–1359, Aug. 2009.
- [41] C. Chakrabortii and H. Litz, "Reducing write amplification in flash by death-time prediction of logical block addresses," in *Proc. 14th ACM Int. Conf. Syst. Storage*, 2021, pp. 1–12.
- [42] M. Cho and D. Kang, "ML-CLOCK: Efficient page cache algorithm based on perceptron-based neural network," *Electronics*, vol. 10, no. 20, p. 2503, 2021.
- [43] L. V. Rodriguez et al., "Learning cache replacement with CACHEUS," in *Proc. USENIX Conf. File Storage Technol. (FAST21)*, 2021, pp. 341–354.
- [44] S. M. A. Iqbal, "Cache-MAB: A reinforcement learning-based hybrid caching scheme in named data networks," *Future Gener. Comput. Syst.*, vol. 147, pp. 163–178, Oct. 2023.



Hui Sun received the Ph.D. degree from Huazhong University of Science and Technology, in 2014. He is an Associate Professor of computer science with Anhui University. His research interests include computer systems, edge computing, performance evaluation, non-volatile memory-based storage systems, file systems, and I/O architectures.



Chen Sun is currently working toward the M.S. degree with Anhui University. His main research interests include SSD storage and intelligent cache management.



Haoqiang Tong is currently working toward the M.S. degree with Anhui University. His main research interests include SSD storage and cache management.



Yinliang Yue received the Ph.D. degree in computer architecture from Huazhong University of Science and Technology, China, in 2011. He is a Professor with the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include computer architecture and storage systems. He has more than 10 publications in major journals and international conferences including the IEEE TRANSACTIONS ON COMPUTERS, ICDCS, IPDPS.



Xiao Qin (Senior Member, IEEE) received the B.S. and M.S. degrees in computer science from Huazhong University of Science and Technology, in 1992 and 1999, respectively. He received the Ph.D. degree in computer science from the University of Nebraska-Lincoln, in 2004. He is currently an Alumni Professor and the Director of Graduate Programs in the Department of Computer Science and Software Engineering, Auburn University. He won an NSF CAREER award in 2009. His research interests include parallel and distributed systems, storage systems, and performance evaluation.