A Machine Learning Based Write Policy for SSD Cache in Cloud Block Storage

Yu Zhang[†], Ke Zhou[†], Ping Huang[†], Hua Wang[†✉], Jianying Hu[‡], Yangtao Wang[†], Yongguang Ji[‡], Bin Cheng[‡]
[†]*Wuhan National Laboratory for Optoelectronics, Key Laboratory of Information Storage System,*
[†]*Engineering Research Center of data storage systems and Technology, Ministry of Education of China,*
[†]*School of Computer Science & Technology, Huazhong University of Science & Technology*
[‡]*Tencent Technology (Shenzhen) Co., Ltd.*
[✉]*Corresponding author: Hua Wang (hwang@hust.edu.cn)*

***Abstract*—Nowadays, SSD cache plays an important role in cloud storage systems. The associated write policy, which enforces an admission control policy regarding filling data into the cache, has a significant impact on the performance of the cache system and the amount of write traffic to SSD caches. Based on our analysis on a typical cloud block storage system, approximately 47.09% writes are write-only, i.e., writes to the blocks which have not been read during a certain time window. Naively writing the write-only data to the SSD cache unnecessarily introduces a large number of harmful writes to the SSD cache without any contribution to cache performance. On the other hand, it is a challenging task to identify and filter out those write-only data in a real-time manner, especially in a cloud environment running changing and diverse workloads.**

In this paper, to alleviate the above cache problem, we propose an ML-WP, Machine Learning Based Write Policy, which reduces write traffic to SSDs by avoiding writing write-only data. The main challenge in this approach is to identify write-only data in a real-time manner. To realize ML-WP and achieve accurate write-only data identification, we use machine learning methods to classify data into two groups (i.e., write-only and normal data). Based on this classification, the write-only data is directly written to backend storage without being cached. Experimental results show that, compared with the industry widely deployed write-back policy, ML-WP decreases write traffic to SSD cache by 41.52%, while improving the hit ratio by 2.61% and reducing the average read latency by 37.52%.

***Index Terms*—cache write policy, cloud block storage, machine learning, SSD**

I. Introduction

Cache server based on solid-state drives (SSDs) is extensively deployed in modern cloud storage systems due to the advantage of random IO performance of SSDs. Compared with traditional hard disk drives, SSDs have the advantageous features of nonvolatility, shock resistance, and energy conservation. Nevertheless, the lifetime of a commonly-used flash-based SSD is limited by the write endurance.

The write policy in a cloud cache system controls how data is populated into the cache and determines which data is evicted from the cache when the cache is full. It has a significant impact on the performance and lifespan of SSD cache. In typical cloud production systems (e.g., CBS [1] and Ceph [2]), all data from write requests are firstly written to the write buffer (made up of DRAM), sequentially recorded in a log file, and then committed to the SSD cache when the amount of dirty data in the write buffer exceeds a threshold. Finally, the written data in SSD will be flushed to the HDD storage asynchronously using write-back policy.

However, after analyzing a one-month long IO trace collected from a typical cloud block storage (CBS) system, we find there are approximately 47.09% data blocks that have only write operations during the month. Deploying either write-back or write-through policy could induce a significant amount of unnecessary write traffic to the SSD cache without enhancing the write performance. On the other hand, simply deploying a write-around policy will inevitably increase the read miss ratio and severely degrade the quality of service.

To solve the above problem, we come up with the idea of distinguishing write-only data from normal data, forming the key component of our proposed novel write policy, a Machine Learning Based Write Policy (denoted as ML-WP). ML-WP can make different treatments for different types of data (i.e., using write-around policy for write-only data and write-through or write-back policy for normal data). The biggest obstacle to implement the idea is to classify these two types of data accurately in real-time. Machine learning technology, which is widely used for data classification and has the ability to cluster different groups of data automatically, can provide a feasible solution to this problem. More details about ML-WP can be found in Section III. Specifically, we make the following contributions in this paper:

- We propose a Machine Learning Based Write Policy (ML-WP), which writes write-only data to the backend store directly and only loads normal data to the SSD caches, so as to prevent the write-only data from entering SSD caches. In doing so, ML-WP decreases a large amount of unnecessary write traffic to SSDs. In the meanwhile, it improves the performance of the cache server due to improved cache utilization.
- To materialize ML-WP and achieve accurate data classification, we employ machine learning methods and exploit several request-level features to cluster data blocks into two groups (i.e., the write-only data and normal data).

This work is supported by the Innovation Group Project of SSS National Natural Science Foundation of China under Grant No.61821003, the National Key Research and Development Program of China under Grant No.2016YFB0800402, and the Innovation Fund of WNLO. Yu Zhang and Ping Huang are the co-first authors.

- We have collected a one-month long IO trace from a cloud block storage system and replayed it in an in-house simulator. The extensive experimental results demonstrate the efficiency of ML-WP.

II. Background

A. Cloud Block Storage and SSD Cache

Cloud block storage (CBS) is a commonly-used cloud architecture with the advantage of flexible scalability and generality. Nowadays, CBS has been deployed and provisioned by the majority of cloud providers, and tenants can access such block-level service like accessing local disks.

To improve the random IO performance of the backend storage servers of CBS, server-end SSD caches are deployed. In such a cloud architecture with SSD caches, requests are first translated to block-level requests. Then, if it is a read request, CBS will try to look up each block in the SSD cache. The data block will be returned from the cache if it is found in the cache space. Otherwise, it will be fetched from the HDD storage. If it is a write request, to improve the write performance, it will first be written to a DRAM write buffer. To ensure data consistency, write requests should be sequentially recorded in a log file before acknowledged to the client because DRAM devices are volatile. Then the dirty data in DRAM are committed to the SSD cache when the number of dirty data in the write buffer exceed a threshold. Finally, the written data in SSD will be flushed to the HDD storage asynchronously once it is replaced by another cache block via using write-back policy.

B. Related Work

1) Write Policies: When there is a write request, the write policy determines whether to load the data to the cache and when to flush the dirty data to the backend storage. Therefore, the write policy has a great impact on the system performance, the data consistency, as well as the write traffic to the cache medium. Summarily, existing write policies either load all the write data to the cache (e.g. write-back and write-through) or never load the write data to the cache but write this kind of data directly to the backing storage (e.g. write-around).

There are existing studies [3]–[6] which focus on optimizing write policy in recent years. Most of them analyze the IO workloads first and then deploy the most appropriate write policy. However, under the changing and diverse cloud workloads, a fixed write policy is destined to be ineffective. In this paper, to overcome the problem, we design a novel write policy tailored for dynamic workloads based on machine learning methods, which can classify different groups of data accurately for the dynamic workloads.

2) Cache Optimization Based on Machine Learning: Cache theory has been developed rapidly over the past few decades [7]. Nowadays, machine learning algorithms have been increasingly deployed for cache optimization because of their capability to cluster dynamic workloads automatically. For example, Perceptron [8] deploys perceptron learning for reuse prediction in the last-level cache. This study designed a single-layer perceptron to integrate various dimension features (e.g., PCs, memory address, reference count, etc.) into a pre-judgment value for cache replacement. Wang et al. [9] have proposed a machine learning based method for one-time-access exclusion, which can accurately identify and filter one-time-accessed photos, to improve the efficiency of the photo cache service for a social network. Flashield [10] employs machine learning methods to select objects which are expected to be read more frequently to be admitted in the SSD cache for cloud key-value workloads, since key-value updates cause serious harmful small random I/O writes to SSD. However, unlike on-chip, file-level or object-level systems, block-level systems have limited information for write-only data identification. In this paper, we have exploited several request-level features to achieve accurate write-only data identification.

III. Design and Implementation

A. Design Overview

In this section, we present the design and implementation of ML-WP. Fig.1 shows the architectural overview of ML-WP. The key point of the ML-WP is the classifier which is used to identify whether the data are normal data or write-only data. Based on the classification result of the classifier, ML-WP writes the write-only data to the backend store directly and only loads the normal data to the SSD caches. In the following, we discuss the details of ML-WP including the machine learning algorithms, feature selection, and the workflow of write process.

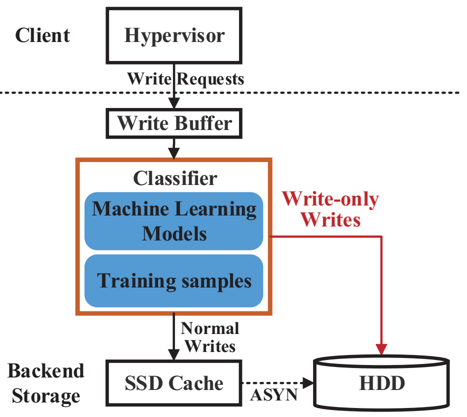


Fig. 1. The overview of the machine learning based write policy.

B. Machine Learning Based Classification

Nowadays, with the rapid application of machine learning technique, it has been increasingly deployed to the computer systems [8], [9], [11], [12]. In this paper, we use supervised machine learning algorithms to distinguish the write-only data from normal data. In this section, we discuss the selection of machine learning algorithms and features.

1) Machine Learning Algorithms: In this paper, we have tested five commonly-used supervised machine learning algorithms, *Naive Bayes*, *Logistic Regression*, *Decision Tree*, *AdaBoost*, and *Random Forest*. More details for these algorithms can be found in [13]. The classification result of different algorithms is shown in Table I. Summarily, *Random Forest* outperforms others in terms of accuracy, but it requires much more time for prediction (averagely $20.23\mu s$ per request). *AdaBoost* is only inferior to *Random Forest* in terms of accuracy, and it also has pretty long prediction time (averagely $14.73\mu s$ per request). *Naive Bayes* represents a good trade-off between accuracy, recall, and prediction time. *Logistic Regression* is only inferior to *Naive Bayes* in terms of recall. *Decision Tree* has the lowest accuracy and recall. The training time of all the algorithms is several hundred milliseconds for training one-day samples.

TABLE I
THE CLASSIFICATION RESULT OF DIFFERENT ALGORITHMS

Algorithm Name	Accuracy (%)	Recall (%)	Training Time(ms)	Prediction Time(μs)
Logistic Regression	83.54	75.21	500.84	1.01
Naive Bayes	83.89	83.91	440.88	1.25
Decision Tree	80.96	68.61	482.62	1.16
AdaBoost	87.85	85.64	462.47	14.73
Random Forest	88.32	82.64	523.68	20.23

2) Feature Selection: Different from previous works on machine learning-based optimization for file-level [9] or on-chip systems [8], deploying machine learning algorithms to block-level systems to classify different data faces great obstacles. The most important point is to select appropriate features from the extremely limited block-level information. The commonly-used information includes temporal features and spatial features (e.g., last access timestamp and address information). In this paper, we expand the feature set by exploiting many request-level features like *average write size*, *write request ratio*, and so on. This is intuitively inspired by the fact that data with similar request-level tends to exhibit similar access characteristics. Specifically, we summarize the block-level features that can be obtained as follows.

- **Temporal features**. These features include access recency and time interval of data blocks. *Last Access Timestamp* is the time point of the last access of a block. In the experiment, we define *Last Access Timestamp* as the time interval of current time and last access time to a block. *Average Re-access Time Difference* represents the time interval of two adjacent accesses of a block. To normalize the time features, we set the time unit to one hour and limit the upper bound for the numeric value to 100 hours.
- **Spatial features**. These features contain the address information (i.e., volume ID, offset, etc.) of the requested data. However, the address keeps changing over time. This will affect the judgment of the algorithm significantly. Therefore, in our implementation, we do not use spatial features.
- **Request-level features**. *Average Request Size* is the average request size whose unit is kilobyte and the upper bound is 100 KB. *Big Request Ratio* is the proportion of requests whose request-size are larger than 64 KB. *Small Request Ratio* is the ratio of requests whose request-size are smaller than 8 KB. *Write Request Ratio* is the ratio of write requests.

Another important issue is the statistic granularity. We know that the minimum granularity is a block (e.g., 8 KB). If we obtain the above features for each block, it will cost too much memory for feature storage which is cost-ineffective. However, a too large granularity will influence the prediction accuracy. In practice, we use a tablet, a 1MB contiguous data area containing multiple blocks, to achieve the trade-off between memory footprint and prediction accuracy. Blocks in the same tablet share the features of this tablet. The **tablet features** are the statistics of the access records of all the blocks in this tablet.

C. ML-WP Workflow

When there is a request from the client, the workflow of the ML-WP proceeds as follows:

(1) All data from the write requests are first written to the memory and sequentially recorded to a log file, and then flushed to the backend storage server when the amount of dirty data reaches a threshold. When there is a data flush from the memory write buffer, go to step 2.
(2) The classifier obtains the tablet features of the currently-flushed data block and predicts its type. If the block is a write-only data, go to step 3. Otherwise, the block is a normal data, execute step 4.
(3) If the data block exists in the SSD cache and the state is dirty, flush the dirty data to the HDD first. The data block will be written to the backend HDD directly, and the process finishes.
(4) The data block will be written to the SSD cache firstly and later flushed to HDD asynchronously using write-back policy.

The classifier model is trained once a day for the next day's prediction. The training samples are collected during the process of the system running. ML-WP adds a sample when there is an eviction for the SSD cache. Specifically, if the evicted-data has one or more read hit, the label of this sample will be set as 0 (i.e., the normal data). Otherwise, the sample will be marked as 1 (i.e., the write-only data). For our purpose, in the real implementation, a write-only data block is a block which has not been read before evicted from the cache. As shown in Table I, the training time of all the algorithms is several hundred milliseconds for training one-day samples and the size of the classifier model is just several megabytes, which represents a negligible effect on the system.

IV. EXPERIMENTAL RESULTS

In this section, we compare ML-WP with write-around policy and write-back policy. We do not test write-through policy because SSDs have the property of nonvolatility and write-back policy is sufficient for ensuring data consistency. As mentioned in III-B, Naive *Bayes* exhibits a good trade-off between accuracy and prediction time compared with other machine learning algorithms in our scenario. Therefore, in the following experiments, we only discuss *ML-WP* based on

Naive Bayes algorithm. We have evaluated the performance of ML-WP policy using an in-house trace-driven simulator [14], which we have made publicly available. In the experiment, the SSD cache size is set to be 10% of the backend HDD storage and the DRAM write buffer size is set to be 0.1% of the backend store. We set the sampling ratio as 0.5%, and the simulation time can be controlled within one day. We run this simulator on an X86-64 computing server with two Intel Xeon E5-2670-v3 CPUs and 128GB DRAM memory.

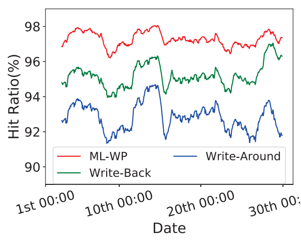
A. IO Traces

To evaluate the efficiency of ML-WP, we have collected a one-month long IO trace from a warehouse of a cloud block storage system [1]. This warehouse has been deployed to serve tens of thousands of cloud disks. The original IO traces contain approximately 200 million records every hour. Using the original data, a simulation for the one-month long trace takes several weeks to finish the simulation. To reduce the time of simulation, we have adopted a novel sampling method, SHARDS, whose effectiveness has been demonstrated in the study [15].

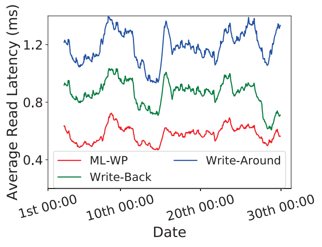
B. Results

1) Read Hit Ratio and Latency: We discuss the read hit ratio and latency because write latency is negligible for tenants (as mentioned above, a write request completes the response to the client as soon as it has been successfully written to the memory buffer and a log file.). Fig. 2 (a) presents the hit ratio results. The x-axis is the date from the first to the last day of a month and the statistical interval is one hour. The y-axis is the hit ratio. To make the curve clearer, all the values are processed using the moving-average method. Averagely, *ML-WP* improves the cache hit ratio by 2.61% and 5.52% compared with the write-back and write-around policy, respectively. Fig. 2 (b) shows the average read latency. The read latency is calculated using Eq. 1. L_{SSD} and L_{HDD} are the read latency of SSDs and HDDs, respectively. In the simulator, we set L_{SSD} and L_{HDD} as 200 μs and 14 ms, correspondingly. On average, *ML-WP* reduces read latency by 37.52% compared with the write-back policy, and by 54.76% compared with the write-around one.

$$\begin{aligned} Latency = Hit\_Ratio \times L_{SSD} + \\ (1 - Hit\_Ratio) \times L_{HDD} \end{aligned} \tag{1}$$



(a)



(b)

Fig. 2. The hit ratio result (a) and the average read latency (b).

2) Data Written to SSDs: Fig. 3 shows the normalized write traffic to SSDs. The write traffic of other policies is normalized to that of write-around policy. Averagely, *ML-WP* reduces write traffic to SSDs by 41.52% (from 7.49 to 4.38) compared with the write-back policy. Write-around achieves the lowest write traffic to SSDs. However, it suffers from, it also exhibits the lowest hit ratio.

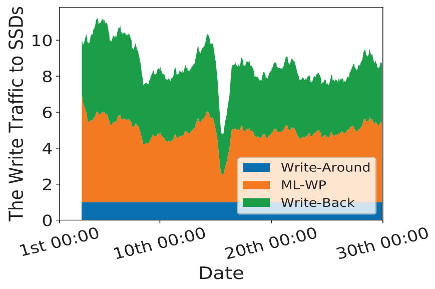


Fig. 3. The write traffic to SSDs.

V. Conclusion

In this paper, we propose a Machine Learning Based Write Policy (ML-WP), which removes a large number of write operations by avoiding writing write-only data into SSD caches. To materialize ML-WP and achieve an accurate data classification, we use machine learning methods and exploit request-level features to cluster data and predict their future reference characteristics. Experimental results demonstrate that, compared with existing write policies, ML-WP decreases a large amount of unnecessary write traffic to SSD, meanwhile, improving the hit ratio of the cache server.

References

[1] Tencent, "Tencent cbs," https://cloud.tencent.com/, 2019.
[2] RedHat, "Ceph," https://ceph.com/, 2019.
[3] D. A. Holland, E. Angelino, and et al., "Flash caching on the storage client," in *Proceedings of ATC*, 2013.
[4] R. Koller, L. Marmol, and et al., "Write policies for host-side flash caches," in *Proceedings of FAST*, 2013.
[5] S. Kim, H. Kim, and et al., "Request-oriented durable write caching for application performance," in *Proceedings of ATC*, 2015.
[6] J. Do, D. Zhang, and et al., "Turbocharging dbms buffer pool using ssds," in *Proceedings of SIGMOD*, 2011.
[7] K. Zhou, Y. Zhang, and et al., "Lea: A lazy eviction algorithm for ssd cache in cloud block storage," in *IEEE 36th ICCD*, 2018.
[8] E. Teran, Wang, and et al., "Perceptron learning for reuse prediction," in *the 49th MICRO*, 2016.
[9] H. Wang, X. Yi, and et al., "Efficient ssd caching by avoiding unnecessary writes using machine learning," in *the 47th ICPP*, 2018.
[10] A. Eisenman, A. Cidon, and et al., "Flashield: a hybrid key-value cache that controls flash write amplification," in *the 16th NSD*, 2019.
[11] Z. Deng, L. Zhang, and et al., "Memory cocktail therapy: a general learning-based framework to optimize dynamic tradeoffs in nvms," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017.
[12] C. Wu, A. Jaleel, and et al., "Ship: Signature-based hit predictor for high performance caching," in *the 44th MICRO*, 2011.
[13] K. P. Murphy, *Machine learning - a probabilistic perspective*, ser. Adaptive computation and machine learning series. MIT Press, 2012.
[14] Y. Zhang, "Mlwp," https://github.com/zydirtyfish/ML-WP, 2019.
[15] C. A. Waldspurger, N. Park, and et al., "Efficient MRC construction with SHARDS," in *Proceedings of the 13th FAST*, 2015.