LPCA: Learned MRC Profiling based Cache Allocation for File Storage Systems

Yibin Gu[1] Yifan Li[1] Hua Wang*[1] Li Liu[1] Ke Zhou[1] Wei Fang[2] Gang Hu[2] Jinhu Liu[2] Zhuo Cheng[2]
[1] Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology
[2] Huawei Data Storage, Huawei Technologies Co., Ltd. *Corresponding Author: Hua Wang
{yibingu,yifan_li_cs,hwang,lillian_hust,zhke}@hust.edu.cn,{fangwei36,hugang27,liujinhu3,chengzhuo}@huawei.com

ABSTRACT

File storage system (FSS) uses multi-caches to accelerate data accesses. Unfortunately, efficient FSS cache allocation remains extremely difficult. First, as the key of cache allocation, existing miss ratio curve (MRC) constructions are limited to LRU. Second, existing techniques are suitable for same-layer caches but not for hierarchical ones.

We present a Learned MRC Profiling based Cache Allocation (LPCA) scheme for FSS. To the best of our knowledge, LPCA is the first to apply machine learning to model MRC under non-LRU, LPCA also explores optimization target for hierarchical caches, in that LPCA can provide universal and efficient cache allocation for FSSs.

KEYWORDS

Cache allocation, Miss ratio curve, Machine learning, Neural network

1 INTRODUCTION

File storage system (FSS) has been a popular storage technique for decades, and it's well-suited to store and organize transactional data or manageable structured data. A file request will be converted to several request streams including dentry cache request, inode cache request, page cache request and mapping table cache request. And FSSs typically employ an unified cache, consisting of multiple caches to accelerate different types of data access streams. Therefore, it is very important to design a good cache allocation strategy [4].

In previous studies, cache space allocation schemes include global sharing policy, static allocation policy and dynamic allocation policy. The sharing policy is simple but may cause performance interference. That is because some aggressive streams may exhaust most of the cache space [10, 14]. In contrast, the static allocation policy statically divides cache resource across streams to deal with performance interference problem. However, this statically partitioning policy may underutilize the valuable cache resources, because the access patterns of workloads keep changing during runtime [7, 18]. To cope with this problem, dynamic allocation policies are proposed, which can obtain near-optimal performance. The mainstream dynamic cache allocation schemes are based on miss ratio curve (MRC). However, we find that those dynamic cache allocation policies cannot work well for the FSS, because the existing cache modeling methods are dependent on LRU replacement algorithm and their applicative cache structure is non-hierarchical [2, 3, 20]. In fact, non-LRU replacement algorithms and hierarchical cache are common in storage systems, such as NAS system [19].

Existing MRC modeling methods can be divided into two groups, the cache modeling based reuse distance method and simulation method. Cache modeling based reuse distance method was first introduced by Mattson [8] et al in 1970. To calculate the reuse distance, Mattson uses a list-based stack which takes $O(NM)$ time complexity and $O(M)$ space complexity for a trace of length N containing M unique references. Because of the high time and space complexity, there are some researches focused on how to optimize calculation performance of reuse distance such as using various data structures, like trees [1, 11], Counter Stacks [17] and OSCA [20]. However, prior methods are based on LRU cache and not suitable for other complex algorithms (e.g. ARC [9], 2Q [6], etc.) which are widely used in practical system. Simulation method was comprehensively evaluated by Mini-Sim [15]. Mini-Sim can emulate a cache with any size and any replacement algorithm by scaling down both the actual cache size and its input data stream which inevitably will incur non-negligible overhead, because the sampling ratio will be limited by accuracy. In general, for non-LRU algorithms used in practice, the MRC built by the existing LRU-tailored reuse distance method lacks precision, and simulation method will incur a huge computational overhead. As a result, it is urgent to propose a new method to realize accurate and fast MRC construction under non-LRU algorithms.

Besides MRC construction, another important factor to cache allocation is how to model the cache structure, which will affect the definition of optimization target. At previous cache allocation scenarios, such as cloud block storage system [20], in-memory key-value cache [2], Last Level Cache (LLC) [3], etc, they just considered caches in the same-layer that are no longer suitable for hierarchical caches in the FSS.

In this paper, we present LPCA, a Learned MRC Profiling based Cache Allocation scheme for FSS, maximizing the read hit ratios and reducing the total number of misses. To achieve this goal, several approaches are proposed. Firstly, LPCA exploits the features of data streams and uses machine learning techniques to construct MRCs. Secondly, LPCA models cache hierarchy, defines the optimization target, and solves it to minimizing the total number of misses. Thirdly, LPCA allocates appropriate cache space for each cache

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530662

accordingly. Specifically, this paper contributes in the following aspects:

(1) We transform the MRC construction problem into a machine learning problem. By learning the impact of the features of the data stream on the MRC, we build a multilayer perceptron (MLP) model for the workload which can accurately and universally construct non-LRU MRC. To the best of our knowledge, we are the *first* to apply machine learning to model non-LRU MRC.
(2) We propose an efficient cache allocation mechanism for FSS, named LPCA, taking the MRC as guides to allocate cache space for multiple data streams, so as to improving the overall performance.
(3) We implement a trace-based simulator and evaluate the effectiveness of our proposed design.

The rest of the paper is organized as follows. Section 2 provides the background information and motivation. Section 3 presents the proposed LPCA method. Section 4 evaluates the performance of LPCA. Section 5 gives the concluding remarks and future work.

2 BACKGROUND AND MOTIVATION

2.1 Cache in File Storage System

To provide user with a simple, durable, flexible and centralized file collaboration, file storage system (FSS) has been developed and deployed extensively. A representative FSS is made up of interface, unified cache and storage pool (as shown in Figure 1). The role of the interface is to translate file requests into several concrete requests flowing inside the FSS. Storage pool employs hard disk drives to provide physical storage for data and metadata, which alone are often unable to meet performance demands. Thus, a FSS typically employs an unified cache to improve performance. The unified cache is composed of a dentry cache, an inode cache, a page cache, and a mapping table cache. Dentry cache caches directory tree. Inode cache stores the meta information of files, such as the creator of the file, the creation date of the file, the size of the file, etc. Page cache caches file objects. Mapping table cache contains the mapping from logical pages to physical pages stored in the storage pool.

When a file request arrives at FSS, it will be converted into three requests, namely dentry, inode and page by the interface. The system will first access the data in the cache, then access the storage pool if any cache missed. One access to the storage takes two times, because we need obtain the index of data before accessing it. And the mapping table cache can reduce the index accesses from the storage pool. Therefore, there is a natural hierarchy between mapping table cache and the other caches.

To improve cache performance, different replacement algorithms (e.g. ARC [9], 2Q [6], FIFO, etc.) are applied in different caches according to their respective access characteristics. That is different from other scenarios which are supposed to only use LRU algorithm.

Previous works have shown that appropriate cache space allocation strategy can improve performance [2, 3, 20]. They often take three steps. First, it models the cache usually in the form of MRC. Second, it defines and solves the optimization target by establishing the relationships between caches. Third, it performs cache reassignment. However, we find that those schemes for cache space allocation cannot work well for the FSS, because the previous cache modeling methods are dependent on LRU and their cache structure is non-hierarchical.

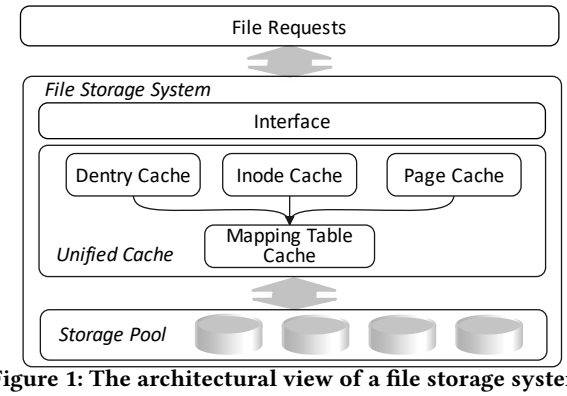


Figure 1: The architectural view of a file storage system.

2.2 The Cache Hierarchy

The cache hierarchy of our FSS is embodied mainly in the relationship between the mapping table cache and other caches. The function of mapping table cache is to reduce the access to the storage pool when cache misses. The specific I/O flow is shown in Figure 2, in which we do not show the process of converting a file request into dengtry, inode and page. For a cache hit, that is ①② and ③: The system searches dentry, inode, page cache in sequence, returns the result and the process finishes. The I/O flow is the same for dentry cache, inode cache and page cache misses. For clear illustration, we show it on the right side of the dashed line in Figure 2. Taking a dentry cache miss as an example: ④When dentry cache misses, the system will look for the index of dentry in the mapping table cache. ⑤If mapping table cache hits, the system can directly access dentry data from the storage pool according to the index. ⑥Otherwise, the system must access index and dentry successively in the storage pool, which involves two back-end accesses.

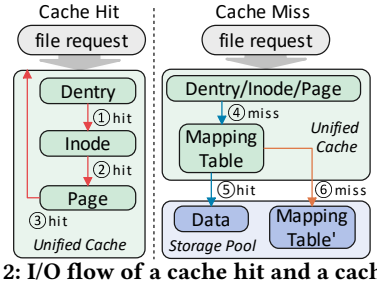


Figure 2: I/O flow of a cache hit and a cache miss.

Due to the special function of the mapping table cache, the number of back-end accesses is not proportional to the number of cache misses observed from the high-level application. Thus, the conventional miss ratio can not truly reflect the impact of missing. In this paper, we introduce a new metric, logic miss ratio (LMR), to express the impact of missing:

$$LMR = \frac{\alpha MN_{MP} + (1-\alpha)(MN_D + MN_I + MN_P)}{N_D + N_I + N_P} \quad (1)$$

Where MN is number of misses and N is the access of cache. And D, I, P, MP represent dentry, inode, page and mapping table cache respectively. $\alpha \in [0, 1]$ is a weight of the mapping table cache, which describes the relation between mapping table cache and the other three caches. The value of α is determined by the impact

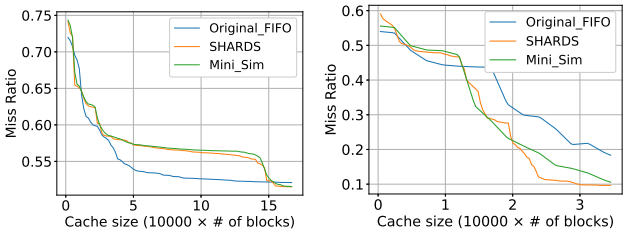
(e.g. latency) of mapping table cache and other cache access to the back-end. If $\alpha = 0$, then the mapping table does not work and the logical miss rate degenerates to the conventional miss rate.

2.3 Comparison of Existing Cache Modeling Methods

The most important thing for an effective dynamic cache space allocation scheme is to model cache by building the low-overhead and accurate MRCs [20]. Thus, We compare the existing commonly-used cache modeling methods and evaluate if they are appropriate for FSS's cache allocation:

- SHARDS [16] is a hash-based spatial sampling technique for reuse distance analysis.
- Mini-Sim [15] can emulate a cache with any size and replacement algorithm by scaling down both the actual cache size and its input data stream.

We obtain publicly-available sub-traces selected from a week-long traces of MSR [13]. It is noteworthy that we only use the prior 600,000 I/Os to construct the MRC under FIFO replacement Algorithm. For SHARD and Mini-Sim, we use random sampling at the rate 0.6 satisfying I/O and accuracy requirements [16].



(a) usr (b) rsrch

Figure 3: MRCs of the MSR traces.

Table 1: The comparison of MRC techniques

	MAE (%)	Runtime (s)
Original	-	741.49
SHARDS	5.41	17.74
Mini-Sim	4.79	248.00

We only show 2 MRCs profiled by different techniques in Figure 3 due to limited space. In Figure 3, original FIFO curve denotes the accurate MRC, the other curves represent approximate MRC construction using SHARD and Mini-Sim respectively. Obviously, there is a huge error (up to 5.41%) between approximate MRC and real non-LRU MRC. The error comes from two aspects: SHARDS is based on LRU, and SHARDS and Mini-Sim have sampling error [16]. A more comprehensive comparison is shown in Table 1, including the MAE and time cost (values are the average of the MSR traces). We can see that these MRC modeling methods are not only unable to accurately build non-LRU MRC but also have an unacceptable time overhead.

This motivates us to design a dynamic cache space allocation strategy that is based on universal (for LRU & non-LUR) and accurate MRC construction. Meanwhile, the strategy should take into account the cache hierarchy in the FSS, whose reason is analyzed in Section 2.2.

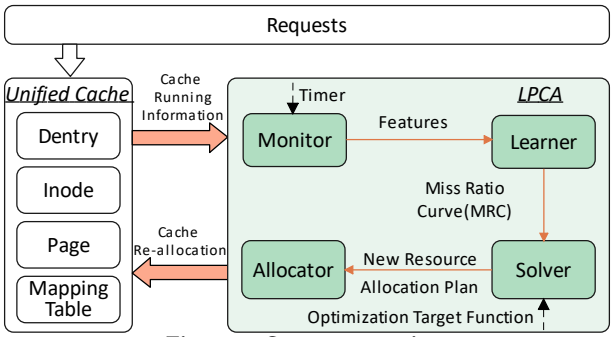


Figure 4: System overview

3 LPCA DESIGN

3.1 Overview

Figure 4 shows the design of LPCA, which includes several components: monitor, learner, solver, and allocator.

Monitor receives I/O requests continuously from the cache and extracts features from the trace.

Learner extract features from the access pattern of a workload and regards the actual MRC as the label. Based on the massive features-label pairs, LPCA gets the ability to train a model for constructing the MRC.

Solver explores the allocation strategy according to the generated MRC of each cache to meet the optimization target function.

Allocator performs space allocation according to the allocation plan.

3.2 Monitor

The existing cache replacement policies, such as LFU, 2Q [6] and ARC [9], etc, are designed to exploit the recency and frequency of accesses. Therefore, in order to correctly construct MRC, monitor focuses on 4 features to extract those information from the trace, which are described as follows:

- **Re-access ratio.** A ratio of the reaccess traffic to the total traffic.
- **Frequency.** The number of total accesses to a same piece of data in the full trace.
- **Frequency-class.** The number of unique frequency (data element accessed times). It represents the class of frequency.
- **Reuse time (RT)** [5]. The number of *total* blocks between two consecutive accesses to the same piece of data. This is also been referred to as gap distance.

Time	0	1	2	3	4	5	6	7	8	9
Address	a	a	b	c	b	d	a	b	d	c
RT	∞	0	∞	∞	1	∞	4	2	2	5

Figure 5: How data features are generated.

Example. Figure 5 gives an example to show how we compute these features. The input is a trace listed in the second row. We use letters to denote addresses. Look at the access to "a" at time point 6. Its reuse time is 4. We defined the reuse time of the first access to data is ∞. After counting the access times of all data, the re-access ratio is calculated to be 0.6. The frequency of "a" is 3 and frequency of other data are "b": 3, "c": 2, "d": 2 respectively. Based on frequencies of all data, we can get the frequency-class of frequency which are "2": 2, "3": 2.

In order to better train the model, it is important that features need be normalized before feeding to the learner. We will evenly divide the cache space into 100 values and record the proportion of RT in the corresponding interval, denoted as $rt[] = [rt_i], i \in [1, 100]$. We record proportion of the frequency of the top n percent data in all data where the values of n are 0.05%, 0.1%, 0.2%, 0.5%, 1%, 2%, 5%, 10%, 20%, 30%, denoted as $freq[]$. Similar to frequency, we use $fclass[]$ to record proportion of the frequency class of the top n where n are 1, 2, 3, 10, 100. And the re-access ratio is denoted as rar. Combining all features, we can obtain the normalized feature vectors $[rar, rt[], freq[], fclass[]] \in \mathbb{R}^{1x116}$.

3.3 Learner

The main function of learner is to construct the MRCs for multiple caches based on features obtained from the monitor. Once MRCs obtained, solver can explore the optimal cache allocation policy. The neural network is a widely used machine learning model. Although the neural network cannot construct the exact MRC, it can approximate the MRC to a highly accurate level (MAE less than 0.01) providing the training set is large enough. In addition, once the model is trained, we can quickly calculate the results. Thus, we use a neural network method to construct the MRC and use a large amount of data to train the model.

The neural network used in LPCA is a 3-layer multilayer perceptron (MLP), as shown in the Figure 6. There are 120 neurons in hidden layer to connect with the input layer and the output layer. For each cache, the input of the MLP includes 4 features described in Section 3.2. The output of this MLP is the MRC expressed by the miss ratios on 100 different cache sizes that evenly divide the cache space. LPCA uses the default mean-square error (MSE) as loss function. Refined parameter selection may lead to better learning effect, but we consider that a more in-depth discussion on parameter optimization is not the focus of this work.

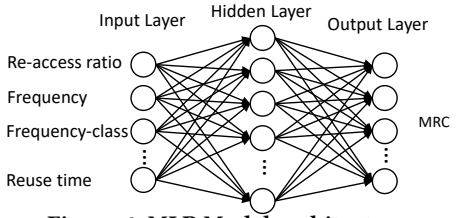


Figure 6: MLP Model architecture

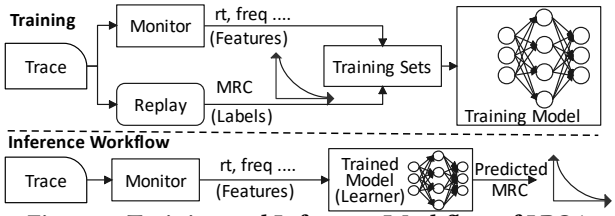


Figure 7: Training and Inference Workflow of LPCA

In order to construct the MRCs accurately, learner needs a large amount of data to train the model. We take features described in the previous subsection as input, and then use the real MRC as a label for training. We use the "replay" method to simulate cache to get the real MRCs. The training and inference workflow of LPCA are shown in the Figure 7. The training phase of the model is offline without affecting the performance of the FSS.

Algorithm 1 Cache allocation algorithm

Data: T: time for collecting features; t: the current time; D, I, P, M : dentry, inode, page and mapping table cache; N_i: the total accesses of cache i.
Input: A sequence of request accesses
Output: Cache space allocation strategy

```
if t<T then
    features, N = Monitor(requests)
else
    for i in (D, I, P, M) do
        features_i, N_i = get_features(i)
        MRC_i = Learner(features_i)
    end for
    strategy = Solver(MRC_i, N_i)
    update(T)
end if
return strategy
```

3.4 Solver

Three cache models, CM_i, $i \in (D, I, P)$, of the first layer for FSS are generated as a function of cache miss rate $MR_i(c_i)$ (where c_i is the target cache size), and total accesses of cache N_i:

$$CM_i = N_i * MR_i(c_i) \tag{2}$$

Based on Eq. 1, we set α to 1/2, which means that the mapping table cache has the same impact as the others. And then, we derive a weight coefficient β from Eq. 1 to model the impact of the mapping table cache on the unified cache. As shown in Eq. 3, β has values between 0 and 1:

$$\beta = (1 + MR_M(c_M))/2 \tag{3}$$

LPCA abstracts cache space allocation as a constrained optimization problem. Its solver explores the space allocation solution that minimize the overall miss number. More specifically, optimization target is shown in Eq. 4.

$$\begin{aligned} min \quad & \sum_{i=x}^{x \in (D,I,P)} p_i * \beta * N_i * MR_i(c_i) \\ s.t. \quad & \sum_{i=x}^{x \in (D,I,P,M)} c_i \le C \end{aligned} \tag{4}$$

Where p_i is a priority for the dentry, inode and page cache, we all set 1 in our experiments. And N_i is the total accesses of cache i, D, I, P, M represent dentry, inode, page and mapping table cache respectively. Our final goal is to minimize the total logical miss number which also equates to maximizing the logic hit rate.

Since optimizing for multiple caches is an NP-Hard problem, we use an existing heuristic algorithm, hill-climbing [12], for our optimization. The pseudocode for cache allocation algorithm is shown in Algorithm 1.

3.5 Allocator

Through the solver, we can get the *targetSize* that the cache instance will be allocated. If the actual size currently occupied by the cache, *actualSize*, is greater than *targetSize*, then certain amount of data items at the head of the to-be-evicted queue will be evicted until *actualSize* equals *targetSize*.

3.6 Discussion

Model universality. As mentioned in Section 3.3, we present a MLP model to construct MRC under non-LRU. In fact, our model is universal, and it is also suitable for building MRC under LRU.

Overhead. The features used in the LPCA are simple to obtain, and the average computational cost of MRC is $0.517ms$ which is controlled within $1ms$. The training of the model is off-line, and the calculation of the model is only carried out one time during each period of space adjustment, so the delay is extremely small and can be ignored.

4 EVALUATION

In this section, we provide comprehensive experiments to evaluate the effectiveness of the LPCA. First, we describe the experimental setup used in this paper. Next, we evaluated the accuracy of the learned MRCs. Finally, we compare the overall efficacy of LPCA in terms of hit ratio and miss reduction.

4.1 Experimental Setup

The workloads. We use MSR Cambridge Traces which are the 1-week block I/O traces of enterprise servers at Microsoft Research Cambridge from SNIA IOTTA repository [13]. We use a 4KB cache block size, so that a cache miss reads from primary storage in aligned, fixed-size 4KB units. The trace combinations used in the MRC accuracy experiment are shown in the Table 2. We use the same way of constructing a learning model as in "diverse workload validation" to construct the learning model in cache allocation experiment, which means that the training set and the test set are different.

Schemes for comparison. In this section, we compare the LPCA with other two methods, including existing static even-allocation method (*Default*) and reuse distance calculated by splay tree with sampling from SHARDS [16] (*SHARDS*).

Simulator design. We designed a trace-based simulator in C++ to perform a quick verification. Similar to the system overview in Figure 4, the simulator is composed of *IO Generator*, *Unified Cache*, *Monitor*, *Learner*, *Solver*, and *Allocator*. The *IO Generator* is used for trace reading and converting trace records into simulator-supported I/O structures. And the functions of the other modules are the same as those described in Section 3. The replacement algorithms used in dentry, inode, page and mapping table cache are FIFO, FIFO, 2Q and FIFO respectively, which refers to a classic FSS.

4.2 MRC Accuracy

In this section we analyze the accuracy of MRCs constructed using LPCA by comparing them to corresponding exact MRCs and other MRC algorithms. First, we preprocess the dataset as following: We segment the trace according to the fixed size which is 40w I/Os in our experiments. The former 90% of sub-traces are used for training the LPCA, and the remaining 10% are used to verify the correctness of the model.

To validate LPCA thoroughly, we designed three groups of experiments as follows:

- **Single Workload Validation** uses the same trace as training set and test set, which used to simulate a single flow workload.

Table 2: The combination of traces used in the experiment

Item	Training Set	Test Set	Item	Training Set	Test Set
1	hm_0	hm_0	5	stg_0	stg_0
2	mds_0	mds_0	6	ts_0	ts_0
3	rsrch_0	rsrch_0	7	wdev_0	wdev_0
4	src2_0	src2_0	8	web_0	web_0

Item	Training Set	Test Set
9	hm_0,mds_0	hm_0,mds_0
10	mds_0,rsrch_0	mds_0,rsrch_0
11	rsrch_0,src2_0	rsrch_0,src2_0
12	src2_0,stg_0	src2_0,stg_0
13	wdev_0,web_0	wdev_0,web_0

Item	Training Set	Test Set
14	hm_0,wdev_0	stg_0
15	src2_0,stg_0	web_0
16	stg_0,wdev_0	mds_0
17	ts_0,wdev_0	stg_0
18	wdev_0,web_0	rsrch_0

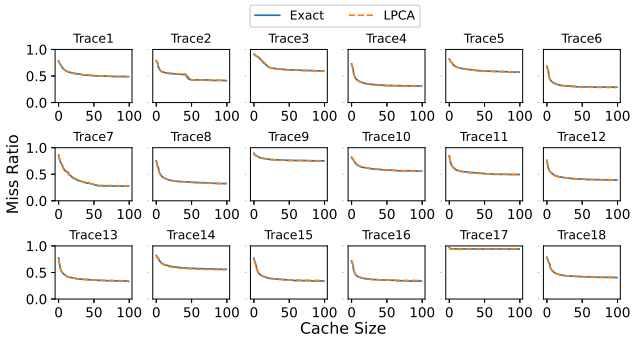


Figure 8: MRCs.

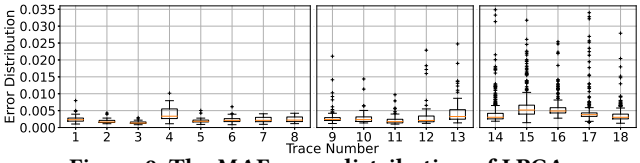
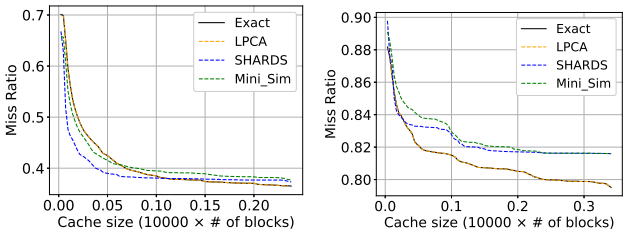


Figure 9: The MAE error distribution of LPCA.

- **Multiple Workload Validation** uses the same multiple traces as training set and test set, which used to simulate a mixed-flow workload.
- **Diverse Workload Validation** uses the different multiple traces as training set and test set, which used to simulate the scenario where a new workload appears.

The traces we used in single, multiple and diverse workload validation are shown in three subforms of Table 2. We use the mean absolute error (MAE) between the approximate and exact MRCs across several different cache sizes to show the accuracy of our MRC modeling methods.

4.2.1 Miss Ratio Curves. Three rows of graphs in Figure 8 plot the predicted MRCs with exact MRCs for single, multiple and diverse workload validation respectively. Each subgraph shows the MRC composed of a subset of each test trace. The x-axis represents 100 different cache sizes, of which the maximum value is 10M. We can see that although the MRCs exhibit drastically different forms, LPCA can always almost accurately predict them. Figure 9



(a) mds_0 (b) ts_0

Figure 10: MRC comparison on mds_0 and ts_0.

shows the MAE metric for single, multiple and diverse workload validation in left-to-right. All workload validation in approximate MRCs shows a median MAE of less than 0.01; most exhibit an MAE bounded by 0.04.

4.2.2 Comparison against other algorithms. We next compare LPCA to SHARDS and Mini-Sim. Figure 10 presents the MRC construction results for the mds_0 and ts_0 dataset. We can see that only LPCA can predict the MRCs accurately. Both SHARDS and Mini-Sim are not even close to LPCA's accuracy.

4.3 Cache Allocation Analysis

In this section, we compare the overall efficacy of LPCA in terms of hit ratio and miss reduction using the above mentioned cache models, respectively. As shown in Figure 11, LPCA can outperform the original assignment policy in the cache hit ratio up to 8.1% and reduces the number of misses up to 24.06%. In terms of hit ratio and miss reduction, LPCA is obviously better than SHARDS.

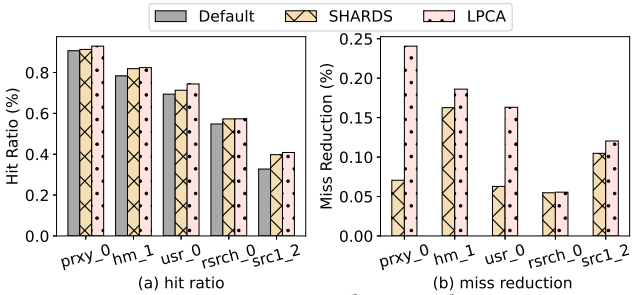


Figure 11: Hit ratio and miss reduction.

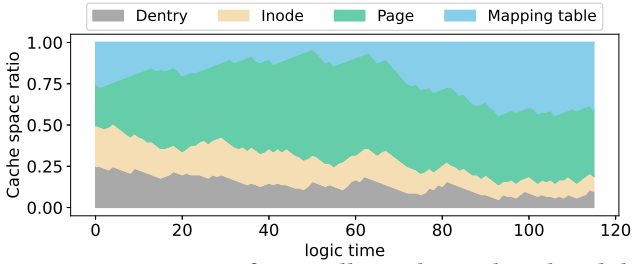


Figure 12: Percentage of space allocated to each cache while hm_1 is running.

Figure 12 shows the percentage of space for each cache when hm_1 is running. The x-axis takes each adjustment cycle as a logical time point, and the adjustment cycle is 40w I/Os. Page cache with high cache requirements, is allocated more cache space than other caches. Dentry and inode cache have low demand, and only allocated less cache space for a period of time. Figure 12 shows that LPCA can effectively and dynamically allocate the optimal cache space for each cache.

5 CONCLUSION AND FUTURE WORK

File storage systems (FSSs) employ unified cache, consisting of multiple caches to accelerate different data access. Most existing cache management policies fall short of being applied to FSS due to their low accuracy and high overhead. In this paper, we propose a learned MRC profiling based cache allocation scheme named LPCA. LPCA exploits the workload features and novelly learns MRCs, meanwhile, profiles the relationship across multiple caches to build an optimization framework that minimizes back-end I/O. We have presented how LPCA can improve cache hit rates for a wide range of workloads by agilely constructing the MRCs. We also find that there is a close relationship between the maximum reduction of cache space together with the adjustment frequency and the adjustment effect, we will leave this to the future study.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (Grant No.62172180), the Innovation Group Project of National Natural Science Foundation of China (Grant No.61821003), and the National Key Research and Development Program of China (Grant No.2019YFB1804300).

REFERENCES

[1] George Almási et al. 2002. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on Memory system performance*. 37–43.
[2] Asaf Cidon et al. 2017. Memshare: a Dynamic Multi-tenant Key-value Cache. In *ATC 17*. USENIX Association, Santa Clara, CA, 321–334.
[3] Nosayba El-Sayed et al. 2018. KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores. In *2018 HPCA*. 104–117.
[4] Ajay Gulati et al. 2012. Demand Based Hierarchical QoS Using Storage Resource Pools. In *ATC 12*. USENIX Association, Boston, MA, 1–13.
[5] Xiameng Hu et al. 2016. Kinetic Modeling of Data Eviction in Cache. In *ATC 16*. USENIX Association, Denver, CO, 351–364.
[6] Theodore Johnson et al. 1994. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm.. In *VLDB*. 439–450.
[7] Zaoxing Liu et al. 2019. DistCache: Provable Load Balancing for Large-Scale Storage Systems with Distributed Caching. In *FAST 19*. USENIX Association, Boston, MA, 143–157.
[8] R.L. Mattson et al. 1970. Evaluation techniques for storage hierarchies. *IBM Systems Journal* 9, 2 (1970), 78–117.
[9] Nimrod Megiddo et al. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST 03*. USENIX Association, San Francisco, CA.
[10] Sparsh Mittal. 2017. A survey of techniques for cache partitioning in multicore processors. *ACM Computing Surveys (CSUR)* 50, 2 (2017), 1–39.
[11] Qingpeng Niu et al. 2012. PARDA: A Fast Parallel Reuse Distance Analysis Algorithm. In *IPDPS*. 1284–1294.
[12] Stuart J. Russell et al. 2003. Artificial intelligence - a modern approach, 2nd Edition. In *Prentice Hall series in artificial intelligence*.
[13] SNIA. [n. d.]. SNIA iotta repository block I/O traces. http://iotta.snia.org/tracetypes/3.
[14] Jian Tan et al. 2018. On resource pooling and separation for LRU caching. *POMACS* 2, 1 (2018), 1–31.
[15] Carl Waldspurger et al. 2017. Cache Modeling and Optimization using Miniature Simulations. In *ATC 17*. USENIX Association, Santa Clara, CA, 487–498.
[16] Carl A. Waldspurger et al. 2015. Efficient MRC Construction with SHARDS. In *FAST 15*. USENIX Association, Santa Clara, CA, 95–110.
[17] Jake Wires et al. 2014. Characterizing Storage Workloads with Counter Stacks. In *OSDI 14*. USENIX Association, Broomfield, CO, 335–349.
[18] Juncheng Yang et al. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *OSDI 20*. USENIX Association, 191–208.
[19] Y. Yasuda et al. 2003. Concept and evaluation of X-NAS: a highly scalable NAS system. In *MSST 2003*. 219–227.
[20] Yu Zhang et al. 2020. OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems. In *ATC 20*. USENIX Association, 785–798.