OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems

Yu Zhang, *Huazhong University of Science and Technology;* Ping Huang, *Huazhong University of Science and Technology and Temple University;* Ke Zhou and Hua Wang, *Huazhong University of Science and Technology;* Jianying Hu, Yongguang Ji, and Bin Cheng, *Tencent Inc.*

https://www.usenix.org/conference/atc20/presentation/zhang-yu

This paper is included in the Proceedings of the 2020 USENIX Annual Technical Conference.

July 15–17, 2020

978-1-939133-14-4

OSCA: An Online-Model Based Cache Allocation Scheme in Cloud Block Storage Systems

Yu Zhang[†], Ping Huang[†§], Ke Zhou[†*], Hua Wang[†], Jianying Hu[‡], Yongguang Ji[‡], Bin Cheng[‡]
[†]*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Intelligent Cloud Storage Joint Research center of HUST and Tencent*
[§]*Temple University,*[‡]*Tencent Technology (Shenzhen) Co., Ltd.*
[*]*Corresponding author: zhke@hust.edu.cn*
○ *Yu Zhang and Ping Huang are the co-first authors*

Abstract

We propose an Online-Model based Scheme for Cache Allocation for shared cache servers among cloud block storage devices. *OSCA* can find a near-optimal configuration scheme at very low complexity improving the overall efficiency of the cache server. *OSCA* employs three techniques. First, it deploys a novel cache model to obtain a miss ratio curve (MRC) for each storage node in the cloud infrastructure block storage system. Our model uses a low overhead method to obtain data reuse distances from the ratio of re-access traffic to the total traffic within a time window. It then translates the obtained reuse distance distribution into miss ratio curves. Second, knowing the cache requirements of storage nodes, it defines the total hit traffic metric as the optimization target. Third, it searches for a near optimal configuration using a dynamic programming method and performs cache reassignment based on the solution. Experimental results with real-world workloads show that our model achieves a Mean Absolute Error (MAE) comparable to existing state-of-the-art techniques, but we can do without the overheads of trace collection and processing. Due to the improvement of hit ratio, *OSCA* reduces IO traffic to the back-end storage server by 13.2% relative to an equal-allocation-to-all-instances policy with the same amount of cache memory.

1 Introduction

With widespread deployment of the cloud computing paradigm, the number of cloud tenants have significantly increased during the past years. To satisfy the rigorous performance and availability requirements of different tenants, cloud block storage (CBS) systems have been widely deployed by cloud providers (e.g., AWS, Google Cloud, Dropbox, Tencent, etc.). As revealed in previous studies [4, 13, 18, 40], cloud infrastructures typically employ cache servers, consisting of multiple cache instances competing for the same pool of resources. Judiciously designed cache policies play an important role in ensuring the stated service level objectives (SLO).

The currently used even-allocation policy called EAP or equal cache partitioning [41] determines the cache requirements in advance according to the respective subscribed SLOs and then provisions cache resources for each cache instance. However, this static configuration method is often suboptimal for the cloud environment and induces resource wastage, because the cloud I/O workloads are commonly highly-skewed [3, 16, 20].

In this paper, we aim to address the management of cache resources shared by multiple instances of a cloud block storage system. We propose an Online-Model Scheme for dynamic Cache Allocation (OSCA) with miss ration curves (MRC). *OSCA* does not require to separately obtain traces to construct MRCs. *OSCA* searches for a near-optimal configuration scheme at a very low complexity and thus improves the overall effectiveness of cache service. Specifically, the core idea of *OSCA* is three-fold. First, *OSCA* develops an online cache model based on re-access ratio (Section 3.2) to obtain the cache requirements of different storage nodes with low complexity. Second, *OSCA* uses the total hit traffic as the metric to gauge cache efficiency as the optimization target. Third, *OSCA* searches for an optimal configuration using dynamic programming method. Our approach is complementary to the most recent on-line scheme SHARDS [34]. It can achieve a suitable trade-off between computation complexity and space overhead (Section 2.3).

As the key contribution, we propose a Re-Access Ratio based Cache Model (*RAR-CM*) to construct the MRC and calculate the space requirements of each cache instance. Compared with previous models, *RAR-CM* does not need to collect and process traces, which can be expensive in many scenarios. Instead, we shift the cost of processing I/O traces to that of tracking the unique data blocks in a workload (i.e., the working set), and this proves advantageous when the number of unique blocks can be efficiently processed in memory. We experimentally demonstrate the efficacy of *OSCA* using an in-house CBS simulator with I/O traces collected from a CBS production system. We are in the process of releasing those traces to the SNIA IOTTA repository [27].

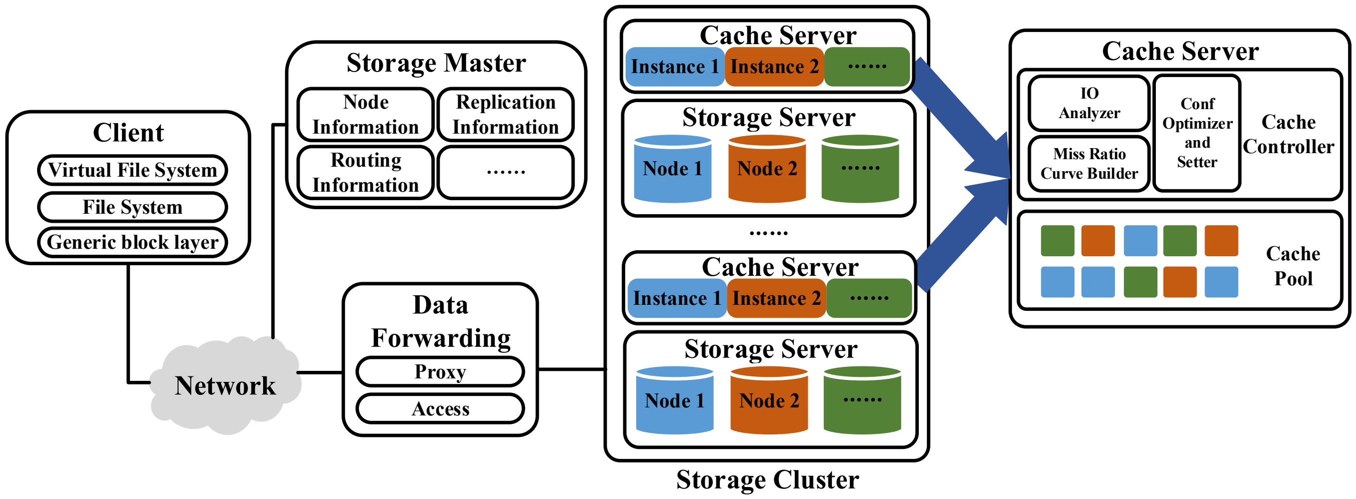


Figure 1: The architectural view of a cloud block storage system (CBS), which includes a client cloud disk layer, Data Forwarding layer, and Storage Cluster containing multiple storage servers each of which is paired with a cache server. The cache server is divided into multiple cache instances respectively responsible for the nodes (i.e., disks) in the corresponding storage server.

The rest of this paper is structured as follows. In Section 2, we introduce the background and motivation of this study and take a detailed look at existing cache modeling methods. In Section 3, we elaborate on the details of our *OSCA* cache management policy. In Section 4, we present our experimental method and the results. In Section 5, we discuss the related work and conclude in Section 6.

2 Background and Motivation

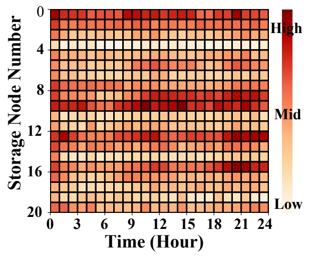
2.1 Cloud Block Storage

To provide tenants with a general, reliable, elastic and scalable block-level storage service, cloud block storage (CBS) has been developed and deployed extensively by the majority of cloud providers. CBS is made up of client layer, data forwarding layer, and storage server layer. The client layer presents tenants with the view of elastic and isolated logic cloud disks allocated according to the tenants' configuration and mounted to the client virtual machines. The data forwarding layer maps and forwards I/O requests from the client-end to the storage server-end. The storage server layer is responsible for providing physical data storage space and it typically employs replication to ensure data reliability and availability. More specifically, a CBS contains multiple components, the client, the storage master, the proxy and access server, and the storage server (as shown in Fig. 1). These components are interconnected through fast fiber-optic networks. The client provides the function of cloud disk virtualization and presents the view of cloud disks to tenants. The storage master (also called the metadata server) assumes the management of node information, replication information, and data routing information. The proxy server is responsible for external and internal storage protocol conversion. In our work, the I/O trace collection tasks are conducted on the proxy server. The access server is responsible for I/O routing that determines which storage node should an access be assigned to based on the MD5 digest calculated from the information of the record. It uses consistent hashing to map each MD5 digest to a positive integer denoting storage node. The storage server consists of multiple failure domains to reduce the probability of correlated failures. Storage servers allocate physical space from conventional hard disk drives, whose performance alone often cannot meet the requirements of cloud applications dominated by random accesses. Therefore, a CBS system typically employs a cache server (comprised of SSDs [18], NVMs [11], or other emerging storage technologies [20]) to improve performance.

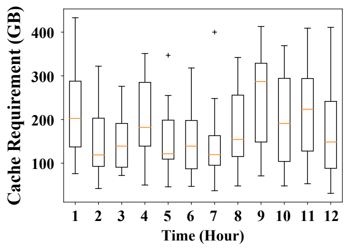
As indicated in Fig. 1, the cache server includes a cache controller and a cache pool. To ensure scalability, there are often multiple cache instances, each associated with one storage node, at the cache server. The user-perceived cloud disk is a collection of logical blocks commonly spread across several physical node disks. A single physical disk is thus shared by multiple virtual disks. As a result, the accesses to a physical disk are mixed patterns. A cache instance is deployed to perform caching for each physical disk and our task is to partition the cache resource among all the cache instances.

2.2 Cache Allocation Scheme

The cache allocation scheme, which is responsible for cache resource assignment, largely influences the efficiency of the cache server. Even-allocation policy (EAP), where each block storage instance receives the same pre-determined amount of cache, is typically used in real production systems for its simplicity. The EAP first analyzes the total cache space requirements in advance according to the defined service-level objectives, and then uniformly allocates cache resources for each cache instance. In essence, it is a static allocation policy and suffers from cache underutilization if over-provisioned and performance degradation if under-provisioned, especially in the cloud environment featuring highly-skewed workloads with unpredictable and irregular dynamics [3, 16, 20]. As shown in Fig. 2 (a), we randomly selected 20 storage nodes and present their IO traffic lasting a period of 24 hours. The figure confirms that the traffic is unevenly distributed to the storage nodes in the realistic CBS production system. Presented from a different perspective, Fig. 2 (b) shows the distribution of cache requirements of those 20 storage nodes during the first 12 hours in order to reach for a level of 95% hit ratio. Again, it shows each storage node has different cache requirements at different times.



(a)



(b)

Figure 2: Fig. (a) presents the frequency of accesses over storage nodes in a typical 24 hour period observed in our traces. The color indicates the intensity of accesses, measured by requests per seconds arriving at each storage node in one-hour time window. The darker the red color in the figure, the more intensive the I/O traffic is. Fig. (b) shows the distribution of cache requirements of those 20 storage nodes during the first 12 hours in order to reach for a level of 95% hit ratio. The orange horizontal line in each box denotes the median cache requirement of the 20 storage nodes, while the bottom and top side of the box represent the quartiles and the lines that extend out of the box (whiskers) represent data outside the upper and lower quartiles.

To improve this policy via ensuring more appropriate cache allocations, there have been proposed two broad categories of solutions. The first category is intuition-based policies such as TCM [19], REF [42], which are **qualitative methods** based on intuition or experience. These policies often provide a feasible solution to the combined optimization problem at an acceptable computation and space cost. For example, according to memory access characteristics, TCM categorizes threads as either latency-sensitive or bandwidth-sensitive and correspondingly prioritizes the latency-sensitive threads over the bandwidth-sensitive threads as far as cache allocation concerns. Such coarse grained qualitative methods are heavily dependent on prior reliable experiences or workload regularities. Therefore, their efficacy is not guaranteed for cloud workloads which are diverse and constantly changing.

The other category is model-based policies, which are **quantitative methods** enabled by cache models typically described by Miss Rate Curves (MRCs), which plot the ratio of cache misses to total references, as a function of cache size [14, 29, 33, 34]. Compared with intuition-based policies, model-based policies are based on cache models containing information about dynamic space requirements of each cache instance and thus are to result in a near-optimal solution. The biggest challenge with quantitative methods lies in constructing accurate miss rate curves at practically acceptable computational and space complexity in an online manner. Most cache models rely on offline analysis due to the enormous computation complexity and space overhead, limiting their practical applicability. A host of research efforts have been conducted to cost-effectively construct miss rate curves with the goal to enable realistic online MRC profiling [4, 29, 31, 33, 34]. Especially, the most recent proposed Spatially Hashed Approximate Reuse Distance Sampling (SHARDS) [34] is an on-line cache model which takes constant space overhead and significantly reduced computational complexity, yet still generating highly accurate MRCs. (Section 2.3 presents more details about SHARDS).

2.3 Existing Cache Modeling Methods

The biggest obstacle to apply an optimal policy to a real system is the huge computational complexity and storage overhead involved to construct accurate cache models which are used to obtain the space requirement of each cache instance. Existing commonly-used cache modeling methods can be divided into two categories, the cache modeling based on locality quantization method and simulation method.

Locality quantization method analyzes the locality characteristics (e.g., Footprint [39], Reuse Distance [34], Average Eviction Time [14], etc.) of workloads and then translates these characteristics into miss ratio curves [7]. The miss ratio curve indicates the miss ratio corresponding to different cache sizes, which can be leveraged to quantitatively determine the cache requirements of different storage nodes. The most commonly used locality characteristic is the Reuse Distance Distribution (as shown in Fig. 3). The reuse distance is the amount of unique data blocks between two consecutive accesses to the same data block. For example, suppose a reference sequence is A-B-C-D-B-D-A, the reuse distance of data block A is 3 because the unique data set between two successive accesses to A is {B, C, D}. The reuse distance is workload-specific and its distribution might change over time.

The distribution of reuse distance has a great influence on the cache hit ratio. More specifically, a data block hits the cache only when its reuse distance is smaller than its eviction distance which is defined as the amount of unique

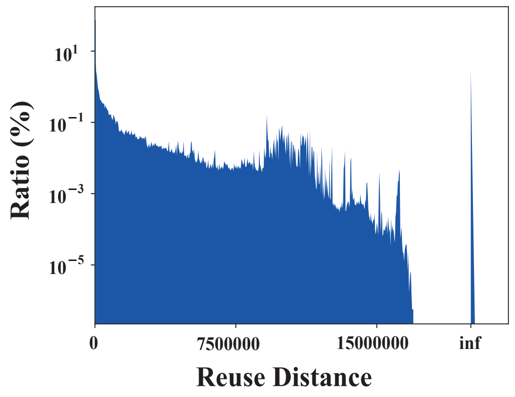


Figure 3: Reuse distance distribution of a one-day long trace from a CBS storage node.

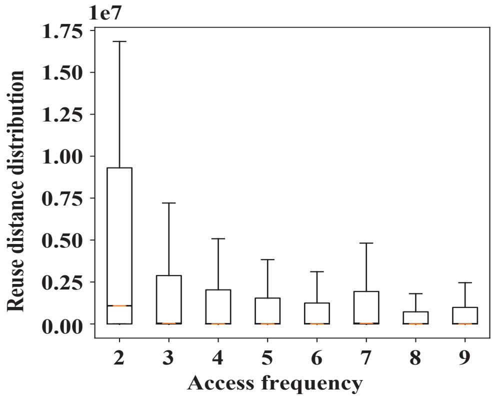


Figure 4: The reuse distance distribution of blocks of a one-day long trace from a CBS storage node, grouped by the access frequencies.

blocks accessed from the time it enters the cache to the time it is evicted from the cache. For a given sequence of block reference, the eviction distance of each block is dependent on the adopted cache algorithm. Different cache algorithms could lead to different eviction distances even for the same block in the reference sequence. The LRU algorithm uses one list and always puts the most recently used data block at the head of the list and only evicts the least recently used block at the tail of the list. As a result, the eviction distance of the most recently used block is equal to the cache size. 2Q [26], ARC [23], and LIRS [17] use two-level LRU lists and a data block can enter the second level lists only when it has been hit in the first level list before. Therefore, these algorithms can result in larger eviction distance for the blocks which have been accessed twice. Similarly, MQ [45] uses multiple-level LRU lists and it causes data blocks with more access frequencies to have larger eviction distances.

In this paper, we focus on modeling LRU algorithm for two reasons. First, LRU is widely deployed in many real cloud caching systems [15, 21]. Second, based on our analysis results of realistic cloud cache, when the cache size becomes larger than a certain size, the advanced algorithms would degenerate to LRU. Fig. 4 presents the reuse distance distribution of blocks with different access frequencies using a one-day long trace from a CBS storage node. The trace is collected from Tencent CBS [30] and we are in the process of making it publicly available via the SNIA IOTTA repository [27]. The bottom and top of each box represent the minimum and maximum reuse distance. The reuse distances of blocks whose access frequencies are larger than 2 are smaller than 0.75×10^7. Therefore, when the cache size becomes larger than 229 GB (0.75×10^7 blocks, each size being 32 KB), the data blocks whose frequencies are larger than 2 can all be hit in the LRU cache because their reuse distances are smaller than the cache size. Other advanced algorithms (e.g., 2Q , ARC, and LIRS) which cause blocks whose occurrences are larger than 2 to have larger eviction distance would degenerate to LRU [44]. Therefore, in our caching system where cache size for each storage node is close to 229 GB (assuming EAP is deployed), the performance differences between LRU and other algorithms are negligible.

Existing cache modeling methods (ours included) calculate the hit ratio of the LRU algorithm as the discrete integral sum of the reuse distance distribution (from zero to the cache size) curve (as shown in Eq. 1).

$$hr(C) = \sum_{x=0}^{C} rdd(x) \tag{1}$$

In the above equation, *hr(C)* is the hit ratio at cache size C and *rdd(x)* denotes the distribution function of reuse distance. However, obtaining the reuse distance distribution has an $O(N * M)$ complexity, where N is the total number of references in the access sequence and M is number of the unique data blocks of references [22]. Recent studies have proposed various ways to decrease the computation complexity to $O(N * log(n))$ using Search Tree [24], Scale Tree [43], Interval Tree [1]. These methods use a balanced tree structure to get a logarithmic search time upon each reference to calculate block reuse distances.

SHARDS [34], further decreases the computation complexity with fixed amount of space. To build MRCs, SHARDS first selects a representative subset of the traces through hashing block addresses. It then inputs the selected traces to a conventional cache model to produce MRCs. Since SHARDS only needs to process a subset of the traces, it significantly reduces the computation overheads and memory space to host the traces. Therefore, SHARDS has the potential to be applied in an on-line manner. All sampled traces can be stored in a given amount of memory by dynamically adjusting the sample ratio. It should be noted that it requires to rescale up the results to obtain the eventual reuse distance for the original traces.

In this paper, we propose an on-line cache model called

RAR-CM to build MRC which is based on a metric called re-access ratio. Our approach does not rely on collecting traces beforehand. Both our approach and SHARDS can be practically applied on-line. Our approach is different from SHARDS in the following aspects. First, SHARDS uses a sampled subset of traces to construct MRCs, while our approach processes I/O requests inline and does not store or process a separate I/O trace. Second, on average it takes $O(lg(M*R))$ asymptotic complexity for SHARDS to update the information in the balanced tree for every sampled block access, where M is the total number of unique blocks in the trace. Our approach only requires to update two counters and thus is $O(1)$.

Table 1 summarizes the comparison between SHARDS and *RAR-CM* in four primary aspects. M, n, and R denotes the total number of unique blocks, the maximum number of records that can be contained in the fixed memory(SHARDS), and the sampling ratio (SHARDS). From the table, we can see that both SHARDS and *RAR-CM* can potentially be applied to construct MRCs in an on-line manner. We can choose to use either of them based on specific scenarios. A general guidance is if we are more concerned about saving computational resources and the available memory can hold support all unique blocks, then our *RAR-CM* is the choice. If we are more constrained by memory and computing resources is not an issue (e.g., we have GPU available), then SHARDS is the choice. In fact, SHARDS and *RAR-CM* are two similar and complementary approaches that can achieve an optimal trade-off point between computation complexity and space overhead. As can be seen from Table 1, one major disadvantage with our approach is that it requires $O(M)$ space to store the information about each unique block. Therefore, in cases where memory is constrained and the working set is relatively large, SHARDS is a better choice.

Table 1: The comparison of *RAR-CM* and SHARDS. M is the number of unique data blocks in the access stream. R denotes the sampling ratio in SHARDS, and n is the number of the sampled unique blocks in the fixed memory. Reuse distribution generation complexity is O(1) for both methods.

	SHARDS	*RAR-CM*
Use full trace	No	Yes
Space Complexity	$O(M*R)$ fixed sample $O(1)$ fixed memory	$O(M)$
Block Access Overhead	$O(log(M*R))$ fixed sample $O(log(n))$ fixed memory	$O(1)$

Simulation-based cache modeling and recently proposed miniature simulation based on the idea of SHARDS [33] need to concurrently run multiple simulation instances to determine the cache hit ratio in different cache sizes. While SHARDS can be applied on-line to process currently sampled traces to obtain the miss ratio curve, the miniature simulation constructs the miss ratio curves based on collected trace beforehand, which could incur no-trivial overhead. We have conducted an experiment with the miniature simulation [33]. Specifically, we run 20 simulation routines (each routine starts 20 threads) simultaneously on a 12-core CPU (i.e., Intel Xeon CPU E5-2670 v3), and this method takes around 69 minutes to analyze a one-day-long IO trace file and most of the time is consumed in trace reading (1.067 µs / record) and IO mapping (2.406 µs / record).

3 Design and Implementation

3.1 Design Overview

OSCA performs three steps, online cache modeling, optimization target defining, and the optimal configuration searching. Fig. 5 illustrates the overall architecture of *OSCA*. Upon receiving a read request from the client, CBS first partitions and routes the request to the storage node and finds the data in the index map of the corresponding cache instance. If it is found in the map on the cache server, the data will be returned to the client directly, and the request will not need to go to the storage server node. Otherwise, the data located in the corresponding physical disk is fetched and returned. A write request is always first written to the cache, and then flushed to the back-end HDD storage asynchronously. All I/O requests are monitored and analyzed by the cache controller for cache modeling. Then the cache controller will find the optimal configuration scheme according to the cache model and the optimization target and finally reassign the cache resource for each cache instance periodically.

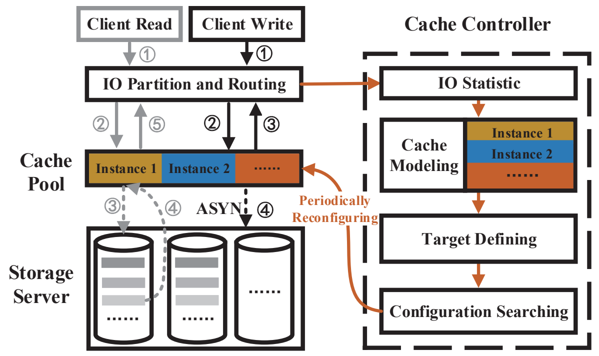


Figure 5: The overall architecture of *OSCA*. Each cache instance is paired with a physical disk which provides storage space for cloud disks. The cache controller monitors the access traffic to physical disks and construct cache models to guide the reassignment of cache resources among cache instances.

3.2 Re-access Ratio Based Cache Model

The main purpose of cache modeling is to obtain the miss ratio curve, which describes the relationship between miss ratio and cache size. The resultant curve can be used in practical applications to instruct cache configurations. We propose a novel online re-access ratio cache model (*RAR-CM*), which can be constructed without the computational overhead of trace collection and processing, when compared with existing cache models. Fig. 6 shows the main components of *RAR-CM*. For a request to block B, we first check its history information in a hash map and obtain its last access timestamp (*lt*) and last access counter (*lc*, a 64-bit number denoting the total number of requests which have been seen so far at the time of last access timestamp, or equivalently the block sequence number of the last reference to block B). We then use *lt*, *lc* and *RAR* curve to calculate the reuse distance of block B. Then the resultant reuse distance is used to calculate the miss ratio curve.

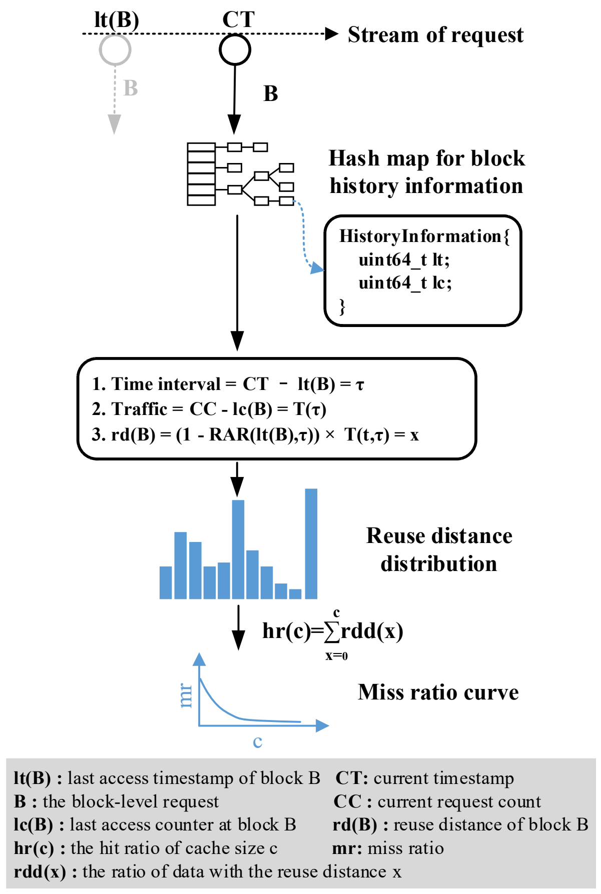

lt(B)
CT
Stream of request
B
B
Hash map for block history information
HistoryInformation{
uint64_t lt;
uint64_t lc;
}
1. Time interval = CT – lt(B) = τ
2. Traffic = CC - lc(B) = T(τ)
3. rd(B) = (1 - RAR(lt(B),τ)) × T(t,τ) = x
Reuse distance distribution
hr(c)=∑_{x=0}^{c} rdd(x)
Miss ratio curve
mr
c
lt(B) : last access timestamp of block B
CT: current timestamp
B : the block-level request
CC : current request count
lc(B) : last access counter at block B
rd(B) : reuse distance of block B
hr(c) : the hit ratio of cache size c
mr: miss ratio
rdd(x) : the ratio of data with the reuse distance x

Figure 6: The overview of re-access ratio based cache modeling. It calculates the reuse distance using re-access ratio and then constructs the miss rate curve based on reuse distance.

RAR, which is defined as the ratio of the re-access traffic to the total traffic during a time interval τ after time t, is expressed as $RAR(t,\tau)$. It essentially represents a metric reflecting how blocks in the following time interval are re-accessed. Fig. 7 shows the re-access ratio during a time interval τ with block access sequence {A, B, C, D, B, D, E, F, B, A}. The number of reaccessed blocks (which includes reaccess to the same block, e.g., B) is 4 (the blue letters marked in Fig. 7), and the total traffic is 10. Therefore, we obtain $RAR(t,\tau)$ = 4 / 10 = 40%.

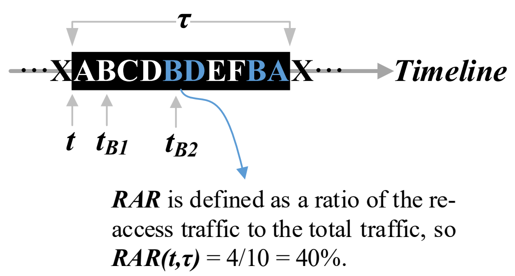

τ
···XABCDBDEFBAX···
Timeline
t
t_B1
t_B2
RAR is defined as a ratio of the re-access traffic to the total traffic, so RAR(t,τ) = 4/10 = 40%.

Figure 7: The definition of re-access ratio of an access sequence during a time period $[t, t+\tau]$.

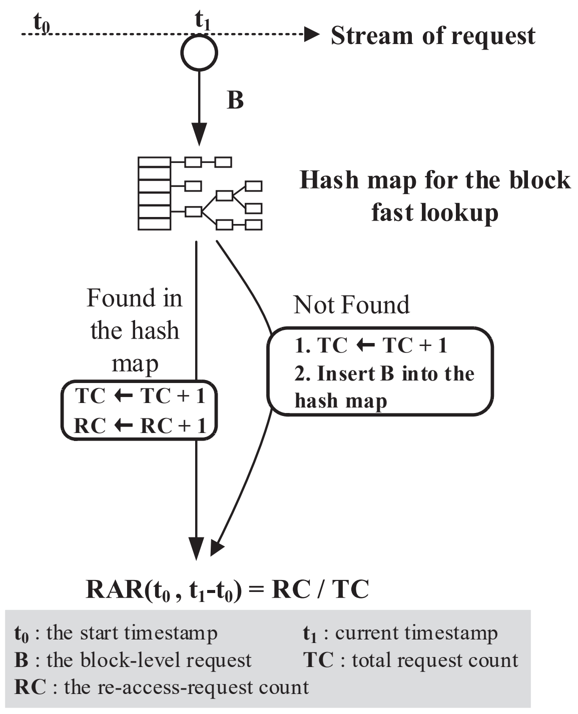
We use the obtained *RAR* for cache modeling because it has a number of favorable properties:

- It can be easily translated to the locality characteristics.
- It can be obtained with low overhead given it's complexity of O(1).
- It can be stored with low overhead of memory footprint.

Locality characteristics. *RAR* can be translated to the commonly used footprint and reuse distance characteristics. As mentioned, the reuse distance is the unique accesses between two consecutive references to the same data block. Assuming that the time interval between two consecutive references of block B is τ, then the reuse distance of block B, $rd(B)$, can be represented by Eq. 2, where $RAR(t,\tau)$ and $T(t,\tau)$ means the re-access ratio and total block accesses between the two consecutive references to block B, respectively. t indicates the last access timestamp of block B. For instance, to calculate the reuse distance of the second B at time t_{B2}, we use $t_{B2} - t_{B1}$ as the τ value for RAR function and 3 as the value of $T(t,\tau)$ in Eq. 2.

$$rd(B) = (1 - RAR(t,\tau)) \times T(t,\tau) \qquad (2)$$

Complexity of O(1). Fig. 8 describes the process of obtaining the re-access ratio curve. *RAR(t_0,t_1-t_0)* is calculated by dividing the re-access-request count (RC) by the total request count (TC) during [t_0,t_1]. To update RC and TC, we first lookup the block request in a hash map to determine whether it is a re-access-request. If found, it is a re-access-request and both TC and RC should be increased by 1. Otherwise, only TC is increased by 1.



t_0
t_1
Stream of request
B
Hash map for the block fast lookup
Found in the hash map
TC ← TC + 1
RC ← RC + 1
Not Found
1. TC ← TC + 1
2. Insert B into the hash map
RAR(t_0 , t_1-t_0) = RC / TC
t_0 : the start timestamp
t_1 : current timestamp
B : the block-level request
TC : total request count
RC : the re-access-request count

Figure 8: The process of obtaining re-access ratio curve. For each incoming block access, it only needs to update two counters, i.e., RC and TC.

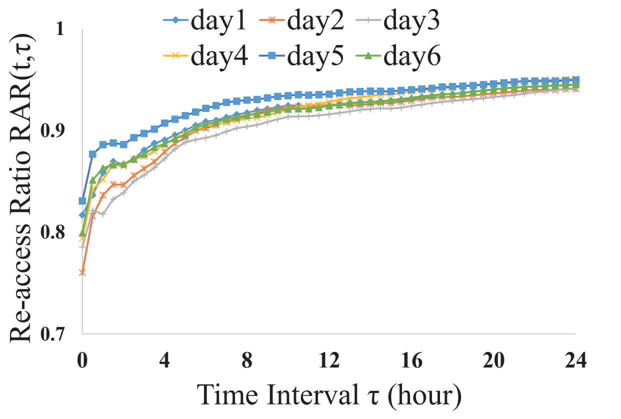


Figure 9: The *RAR* curves of the six days are similar and can be fitting-curved using as logarithmic functions. These *RAR* curves are calculated based on the traces collected from one storage node of Tencent CBS.

Memory footprint. Fig. 9 shows the RAR curves calculated at the end of each of the six trace days. As can be seen, those curves have similar shapes and can be approximated by logarithmic curves which have the form of $RAR(\tau) = a * log(\tau) + b$, where τ is the time variable. Therefore, we only store the two parameters to represent the curve, which has negligible overhead. Note that the presented logarithmic curves are obtained from our traces. Others ways of compactly representing the distribution are possible (e.g., a Weibull distribution [36]). Moreover, for different workloads the shapes of the *RAR* curves may vary and correspondingly we could approach that with other distributions.

In summary, we calculate the *RAR* curve using a hash map to decide whether a block reference is a re-access or not and then based on the *RAR* curve we obtain the reuse distance distribution according to Eq. 2. Finally, the reuse distance distribution is translated to the miss ratio curve leveraging Eq. 1. With the miss ratio curve in place, we then perform cache reconfiguration. Ideally, we want to obtain all the *RAR* curve at each timestamp which is cost-ineffective. Fortunately, we observe that $RAR(t, \tau)$ is relatively insensitive to time t by analyzing a week-long cloud block storage trace (a mixed-trace consisting of tens of thousands of cloud disks' requests). Specifically, although cloud workloads are highly dynamic, we observe that the RAR curves are stable over a couple of days, which means changes of *RAR* curve are negligible over days. Therefore, in our experiment we only calculate the *RAR* curve once a day to represent the *RAR* curve for the next coming day. Specifically, assume the starting time of next day is t_0 and a block is accessed at time t_1. Then we use $t_1 - t_0$ as input to the *RAR* curve function to calculate it's reuse distance using Eq. 2. Note that if the block is accessed the first time, then it's reuse distance is to set to infinitely large, meaning it is a miss.

3.3 Optimization Target

After obtaining cache modeling, we should define a cache efficiency function as the optimization target. Previous studies have suggested a number of different optimization target (e.g. RECU [41], REF [42], et al.). For instance, RECU considers the elastic miss ratio baseline (EMB) and the elastic space baseline (ECB) to balance tenant-level fairness and the overall performance. Considering our case being cloud server-end caches, in this work we use the function E in Eq. 3 as our optimization target. $HitRatio_{node}$ represents the hit rate of the node and $Traffic_{node}$ denotes the I/O traffic to this node. Therefore, this expression represents the overall hit traffic among all nodes. The bigger the value of E is, the less traffic is sent to the backend HDD storage. Admittedly, other optimization targets are also possible and can be decided taking into service level objective account. Based on this target function, our aim is to find a cache assignment method which leads to the largest hit traffic and the smallest traffic to the back-end storage server.

$$E = \sum_{node=1}^{N} HitRatio_{node} \times Traffic_{node} \tag{3}$$

3.4 Searching for Optimal Configuration

Based on the cache modeling and defined target mentioned above, our *OSCA* searches for the optimal configuration scheme. More specifically, the configuration searching process tries to find the optimal combination of cache sizes of each cache instance to get the highest efficiency E.

To speed up the search process, we use dynamic programming (DP), since a large part of calculations are repetitive. A DP method can avoid repeated calculations using a table to store intermediate results and thus reduce the exponential computational complexity to a linear level.

3.5 Implementation Details

Algorithm 1 presents the pseudocode of the process of our *RAR-CM*. The content of block history information is shown in Fig. 6. The re-access ratio curve and the reuse distance distribution are arrays. The subroutine *update_reuse_distance* (Algorithm 2) is used to update the reuse distance distribution *RD* according to the re-access ratio curve *RAR*. And the subroutine *get_miss_ratio_curve* (Algorithm 3) is used to obtain the miss ratio curve according to the reuse distance distribution *RD*. Specifically, *RD* is formed by an array containing 1024 elements, each denoting 1 GB wide (32768 cache blocks of size 32 KB), representing the reuse distances up to 1 TB. The *get_miss_ratio_curve* calculates the cumulative distribution function for *RD*.

From the pseudocode, we can know that the reuse distance calculation of each block is very lightweight which only involves several simple operations and takes hundreds of nanoseconds. This means *RAR-CM* has a negligible influence on the storage server. And the history information of each referenced block contains two 64-bit numbers, occupying very little memory space. More details for the discussion of CPU, memory, network usage can be referenced to Section 4.5.

Algorithm 1: The pseudocode of the *RAR-CM* process

```
Data: Initialize the global variable: hash map for block
      history information $H$, current timestamp $CT$,
      current block sequence number $CC$, and the
      re-reference count $RC$. The re-access ratio curve
      $RAR$. The reuse distance distribution $RD$
Input: a sequence of block accesses
Output: output the miss ratio curve
1  while has unprocessed block access do
2      $B \leftarrow$ next block
3      $CC \leftarrow CC + 1$
4      $CT \leftarrow$ current timestamp
5      if $B$ in $H$ then
6          $RC \leftarrow RC + 1$
7          $RAR(H(B).lt, CT - H(B).lt) = RC/CC$
8          $H(B).lc \leftarrow CC$
9          $H(B).lt \leftarrow CT$
10     end
11     else
12         Initialize H($B$)
13         $H(B).lc \leftarrow CC$
14         $H(B).lt \leftarrow CT$
15         Insert $H(B)$ into $H$
16     end
17     $update\_reuse\_distance(B)$
18 end
19 return $get\_miss\_ratio\_curve(RD)$
```

Algorithm 2: Subroutine *update_reuse_distance*

```
Input: currently accessed block $B$
1  if $B$ in $H$ then
2      $time\_interval = CT - H(B).lt$
3      $traffic = CC - H(B).lc$
4      $rd(B) = (1 - RAR(H(B).lt, time\_interval)) * traffic$
5      $RD(rd(B)) \leftarrow RD(rd(B)) + 1$
6  end
```

Algorithm 3: Subroutine *get_miss_ratio_curve*

```
Input: the reuse distance distribution $RD$
1  $total = sum(RD)$
2  $tmp = 0$
3  for element in $RD$ do
4      $tmp \leftarrow tmp + element$
5      $MRC.append(1 - tmp/total)$
6  end
7  return $MRC$
```

4 Evaluation

4.1 Experimental Setup

Trace Collection. To evaluate *OSCA*, we have collected six-day long I/O traces from a production cloud block storage system using a proxy server which is responsible for I/O forwarding between client and storage server. The cloud block storage system has served tens of thousands of cloud disks. The trace files record every I/O request issued by the tenants and each item of the trace file contains the *request timestamp*, *cloud disk id*, *request offset*, *I/O size*, and so on. To not influence tenants' I/O performance, we have optimized the collection tasks by merging and reporting I/O traces to the trace storage server periodically. We trigger the collection tasks to scan the local I/O logs on the proxy server and report the merged I/O traces every hour, which is an appropriate time interval that can balance the number of tasks with the size of the merged trace files.

Simulator Design. We have implemented a trace-driven simulator in C++ language for the rapid verification of the optimization strategy. The architecture of the simulator consists of an *I/O generator*, an *I/O router*, *cache instances* and *storage nodes*, etc. The *I/O generator* is for trace reading and transforming the trace records to the specific I/O structure of the simulator. The *I/O router* is responsible for request routing and forwarding, which is used to simulate the forwarding layer (shown in Fig. 1) to map each request to a specific storage node. The *storage nodes* simulate the nodes at the storage server layer (shown in Fig. 1). Each node is responsible for one magnetic storage drives and maintains the data mapping relationships inside that node. The *cache instances* is between the *I/O router* and the *storage nodes* and is part of the cache layer of the storage system. Each instance belongs to only one storage node and consists of the index map, metadata list, configuration structure, statistic housekeeping data structure, etc. The index map is implemented by using the *unordered_map* in C++ STL and the metadata list is organized according to the cache algorithm. Considering our cloud simulator is designed to be cloud storage system oriented, we choose only to use our own CBS trace in our evaluations. In our future work, we plan to evaluate our approach using other available traces, especially for comparing the efficacy of constructing MRCs.

4.2 Basic Comparisons

In this section we compare the cache model based on re-access ratio (hereafter called *RAR-CM*) with other three methods, including existing even-allocation method (*Original*), miniature simulation with the sampling idea from SHARDS [33] (*Mini-Simulation*), and an ideal case (*Ideal*) where exact miss ratio curves are used in placement of constructed cache models. We uses the jhash [35] function in implementing *Mini-Simulation* for the uniform randomized spatial sampling. This method leverages jhash to map each I/O record (using attributes like volume ID and data offset) to a location address. The accesses to the same physical block will be hashed to the same value. The I/O record will be selected only when $(V \ mod \ P) < T$, where P and T means the modulus and threshold, respectively. As in SHARDS, $SR = T/P$ represents the sampling ratio. In our experiments, we adopt a fixed sampling ratio of 0.01. We use the *RAR* curves in the prior 12 hours when calculating reuse distance. As illustrated in Fig. 9, the *RAR* curves exhibit good stability, i.e., they show minimum variations in the following days.

Table 2 shows the overall experimental results. In our configuration, we set the average cache size for each storage node as 200 GB (currently-practical configuration). All cache models perform comparably in terms of hit ratio. However, we have observed important back-end traffic savings despite of the seemingly negligible hit ratio improvements. *RAR-CM* compared to *Original* assignment policy with same amount of cache space reduces I/O traffic to back-end storage server by 13.2%. To achieve the same improvement, the *Original* method would require 50% additional cache space on each storage node (i.e., increase from 200 GB / Node to 300 GB / Node) based on the traces we collected from the production CBS system.

Table 2: The overall experimental results

	Hit Ratio	Back-end Traffic	Average Error	Extra Traffic
Original	94.45%	1	-	No
Mini-Simulation	94.85%	0.929	0.017	Yes
RAR-CM	95.14%	0.868	0.005	No
Ideal	95.49%	0.806	0	No

Note: The back-end traffic are normalized to that of *Original* method.

The hit ratio of *Mini-Simulation* is also quite high: 0.29% and 0.64% less than our cache model and the ideal model, respectively. This is consistent with the results in the earlier studies [33].

4.3 Miss Ratio Curves

We next take a closer look at the miss ratio curves of the three cache models. Fig. 10 shows the miss ratio curves of *RAR-CM* (the blue solid line with the cross), *Mini-Simulation* based on SHARDS (the green dotted line), and the exact simulation (the orange solid line). This figure shows the results of 20 randomly selected, but representative storage nodes. Other storage nodes have similar results. The cache space requirements vary among storage nodes and the curves of *RAR-CM* are closer to the curves of the exact simulation than that of *Mini-Simulation* in most cases. The advantage might be attributed to *RAR-CM* constructing the cache model based on the full set of trace and *Mini-Simulation* using spatial sampling causing some fidelity loss.

To evaluate the deviations of curves against the exact miss ratio curves, we report the metric of Mean Absolute Error (MAE) commonly used in evaluating cache models [33, 34]. In our experiments, we compute miss ratio curves at cache sizes 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 200, 300, 400 and 500 GB. Fig. 11 presents the MAE error distributions of *RAR-CM* and *Mini-Simulation* for the selected 20 storage nodes. The MAE averaged across all 20 storage nodes (labeled "Total") for *RAR-CM* is smaller than for Mini-Simulation: 0.005 vs 0.017, in addition to being smaller for each of the 17 out of the 20 nodes.

4.4 Overall Efficacy of OSCA

In this section, we compare the overall efficacy of *OSCA* in terms of hit ratio and backend traffic using the above mentioned three cache models, respectively. We present the results

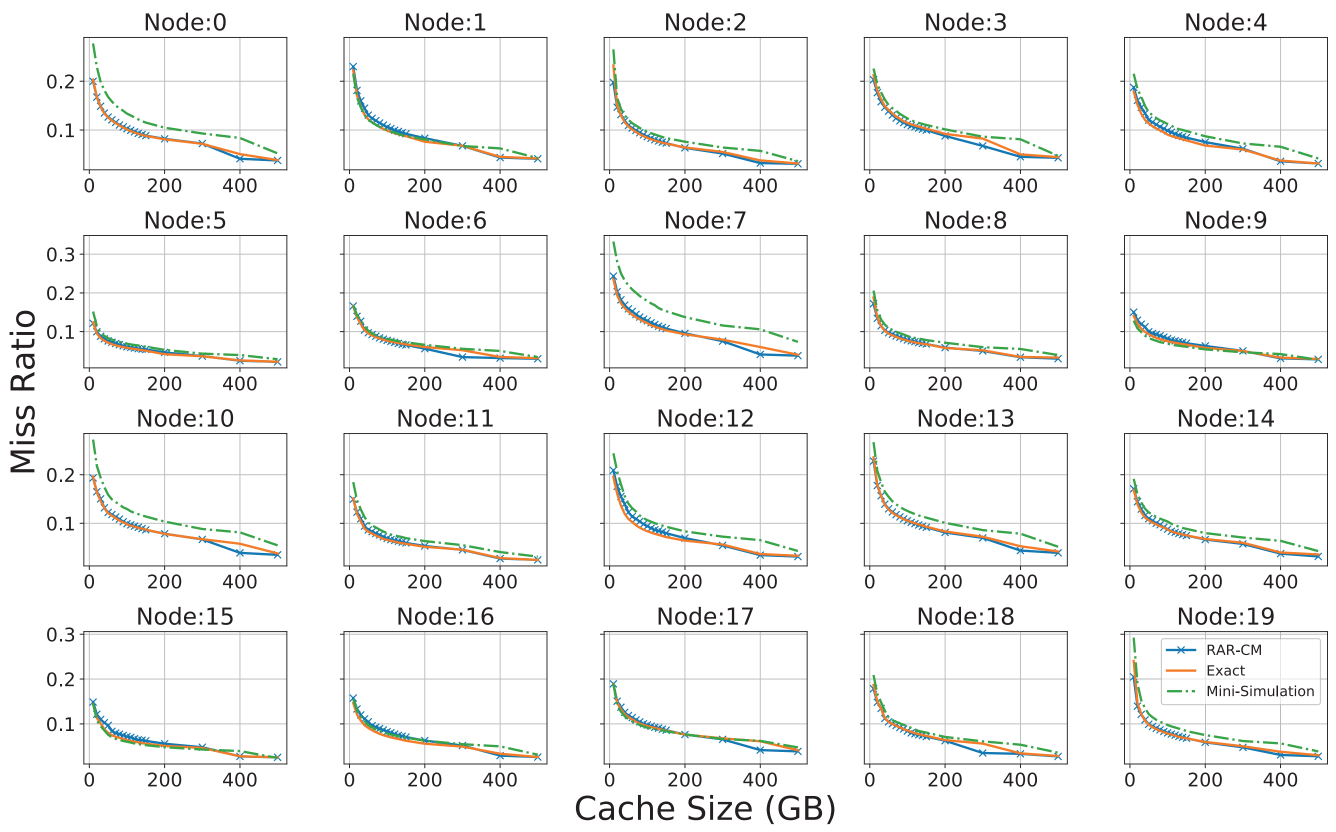


Figure 10: The miss ratio curve of 20 storage nodes. The cache space requirements vary among storage nodes and the curves of *RAR-CM* are closer to the curves of the exact simulation than that of *Mini-Simulation in most cases*.

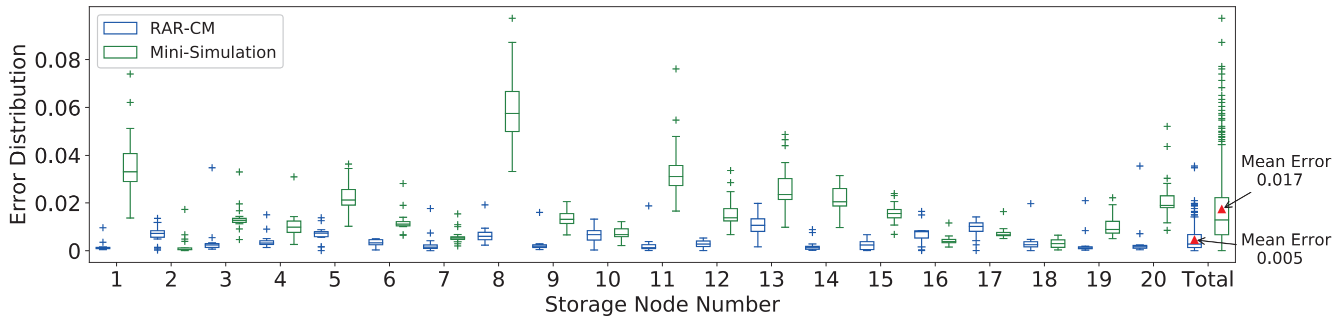
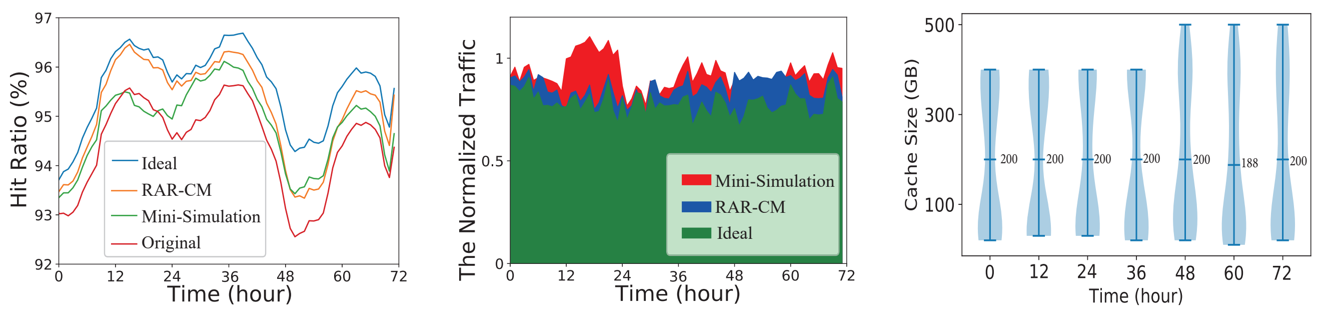


Figure 11: The MAE error distribution of our method *RAR-CM* and *Mini-Simulation* among storage nodes. The last two boxes are total MAE results. The middle lines in boxes indicate the middle values. The bottom and top side of the box represent the quartiles and the lines that extend out of the box (whiskers) represent data outside the upper and lower quartiles.

from the last three days of the trace, using the first 3 days as warm up periods. As shown in Fig. 12-a, *OSCA* based on *RAR-CM* can outperform the original assignment policy in the cache hit ratio without requiring additional cache space. Fig. 12-b shows the back-end traffic with different cache management policies. The back-end traffic is normalized to that of *Original* method. From the figure, we can know that on average, *OSCA* based on *RAR-CM* can reduce I/O traffic to back-end storage server by 13.2%. As shown in Fig. 12, *RAR-CM* results in slightly better hit ratios that *Mini-Simulation* except for hours $48-60$.

Fig. 12-c show the cache size configuration for each node at different times determined by our *OSCA* algorithm with *RAR-CM*. It can be seen that the demand for cache space varies considerably between nodes and our approach did respond correspondingly to meet the needs at different times.



(a) (b) (c)

Figure 12: Fig. (a) and Fig. (b) represents the hit ratio results for the last three days and the normalized back-end traffic using the three cache models, respectively. Fig.(c) shows OCSA adjusts the cache space for 20 storage nodes dynamically in response to their respective cache requirements decided by our cache modeling. The middle line in Fig. (c) represents the average cache size for each node. The results are obtained from traces mentioned in Section 4.1.

Based on the optimal cache size configuration scheme, *OSCA* periodically reassigns the corresponding cache size to each cache node every 12 hours.

4.5 Discussion

When trace collection and processing present a significant cost, *RAR-CM* offers an attractive alternative to other state-ot-the-art techniques. In this section, we make a comparison between *RAR-CM* and *Mini-Simulation* in terms of CPU, memory, network usage.

As mentioned in Section 3.2, upon each block request, *RAR-CM* first checks its history information in a hash map and calculates the block reuse distance. The history information of each referenced block contains two 64-bit numbers denoting the last access timestamp and the block sequence number of the last reference to each block, respectively. In our experiment, there are approximately 55.8 million unique blocks referenced each day in a storage node, occupying only 0.87 GB memory space via using *RAR-CM*. Besides the low memory resource usage, *RAR-CM* does not induce extra network traffic as all the computation is completed on the storage server nodes, enabling the miss ratio curves to be constructed and readily available in an online fashion. As for the CPU resource usage, as shown in Section 3.2, the reuse distance calculation of each block is very lightweight which only involves several simple operations and takes hundreds of nanoseconds.

Mini-Simulation needs to concurrently run multiple simulation instances to construct the cache miss ratio in different cache sizes. However, for very long traces, this method can consume a large number of computation resources (in our implementation, we start a thread in the main routine for each cache algorithm in a specific cache size). More importantly, I/O traces (there are about 4.46 billion I/O records per day in a typical CBS system) ought to be transmitted to and analyzed by a dedicated analysis system to avoid influencing service times. According to our experimental results, the transmission of the I/O records from these 20 nodes consumes approximately 72 GB of network bandwidth each day.

To quantify the runtime overhead, we have experimented with the *Mini-Simulation* algorithm. Specifically, we run 20 simulation routines (each routine starts 20 threads) simultaneously on a 12-core CPU (i.e., Intel Xeon CPU E5-2670 v3). The traces are stored in a storage server and each thread accesses the traces via the network file system. This method takes around 69 minutes to analyze a one-day-long I/O trace file and most of the time is consumed in trace reading (1.067 µs / record) and I/O mapping (2.406 µs / record). The I/O mapping determines which storage node should a record be assigned to based on the MD5 digest from the information of the record. We maintain the total time for the trace reading and I/O mapping and divide them by the total number of records processed to obtain the overhead per record.

5 Related Work

Our work is mostly related to the management of shared cache resource, which widely exists in various contexts, including multi-core processors, web applications, cloud computing and storage. A variety of methods have been proposed and they can be generally classified into heuristic methods, model-based quantitative methods.

Heuristic Methods: To achieve fairness in cache partitioning, the max-min fairness (MMF) and weighted max-min fairness methods are popularly used [12]. These two methods fairly satisfy the minimum requirements of each user and then evenly allocate unused resources to users having additional requirements. Different from MMF, Parihar et al. [25] propose the method of cache rationing, which ensures that the program cache space is not less than a set value and free cache space is allocated to a specific program. Kim, et al. [19] propose TCM which divides threads into delay-sensitive and bandwidth-sensitive groups and apply different cache policies to them. Similar to the TCM method, Zhuravlev et al. [46] proposed a scheduling algorithm called Distributed Intensity (DI), which adjusts the scheduling algorithm by analyzing the classification schemes of each thread through a novel methodology. Other methods, like [32], [12], and [42], have been proposed based on the game theory principles.

Model-based Quantitative Methods: Besides heuristic methods mentioned above, there have also been proposed many quantitative methods. These methods use locality metrics (e.g., Working Set Size, Average Footprint, Reuse Distance, and so on) to quantify the locality of the access patterns so as to predict the hit (or miss) ratio [7]. Reasonably, a shared-cache partition can be efficient using quantitative methods. **Working Set Size**. Inspired by the principle of locality, there are many studies [2, 9, 10] modeling the locality characteristics using working set size (WSS). For instance, based on the WSS theory, Arteaga et al. [2] propose an on-demand cloud cache management method. Specifically, they used Reused Working Set Size (RWSS) model, which only captures data with strong temporal locality, to denote the actual demand of each virtual machine (VM). Using the RWSS model, they can satisfy VM cache demand and slow down the wear-out of flash cache as well. **Footprint**. Footprint, which is defined as the number of unique data blocks referenced in a time interval, has been widely applied to cache resources allocation. Various methods have been proposed to estimate the footprint of workloads [6, 8, 28, 37] and they make trade-off between the complexity and accuracy of the measurement. Xiang et al. [38] propose the HOTL theory, which calculates the average footprint in a linear time complexity and apply the HOTL theory to transfer the average data footprint to reuse distance and predict the miss ratio in their following work [39]. By using this method, they can predict the interference of cache sharing without the need of parallel testing with multiple of cache sizes, and thus the miss ratio can be evaluated with low overhead. **Reuse Distance**. Reuse distance, defined as the unique accesses between two consecutive references to the same data, can be translated to hit ratio and a host of research efforts have been put to efficiently obtain reuse distance. Mattson et al. [22] give the definition of reuse distance and propose a specific method to measure reuse distance. Later researches use tree-based structure to optimize the computation complexity of reuse distance calculation [1, 5, 24, 43]. Waldspurger et al. [34] propose a spatially hashed approximate reuse distance sampling (SHARDS) algorithm to efficiently obtain reuse distance distribution and construct approximate miss rate curve. Hu et al. [14] propose the concept of average eviction time (AET) and relate the miss ratio at cache size c with AET using the formula mr(c) = P(AET(c)), which indicates that the miss ratio is the proportion of data whose reuse distance is greater than AET. In this study, AET is obtained through the Reuse Time Histogram (RTH) with a certain sampling method.

6 Conclusion

Cloud block storage (CBS) systems employ cache servers to improve the performance for cloud applications. Most existing cache management policies fall short of being applied to CBSs due to their high complexity and overhead, especially in the cloud context with large amount of I/O activity. In this paper, we propose a cache allocation scheme named *OSCA* based on a novel cache model leveraging re-access ratio. *OSCA* can search for a near optimal configuration scheme at a very low complexity. We have experimentally verify the efficacy of *OSCA* using trace-driven simulation with I/O traces collected from a production CBS system. Evaluation results show that *OSCA* offers lower MAE and computational and representational complexity compared with miniature simulation based on the main idea of SHARDS. The improvement in hit ratio leads to a reduction of I/O traffic to the back-end storage server by up to 13.2%. We are working on releasing our traces via the SNIA IOTTA repository [27] and integrating our proposed technique into the real CBS product system.

Acknowledgments

We would like to thank the anonymous reviewers for the valuable feedbacks and comments. We are especially grateful to our shepherds Jiri Shindler and Michael Mesnier for their tremendous help in improving the presentation and paper quality. We would also like to thank Tencent Technology (Shenzhen) Co., Ltd. for experimental environment, I/O trace support and releasing the trace to the community. This work is supported by the Innovation Group Project of the National Natural Science Foundation of China No.61821003.

References

[1] George Almási, Călin Caşcaval, and David A Padua. Calculating stack distances efficiently. In *Proceedings of the 2002 workshop on Memory system performance*, pages 37–43, 2002.

[2] Dulcardo Arteaga, Jorge Cabrera, Jing Xu, Swaminathan Sundararaman, and Ming Zhao. CloudCache: On-demand flash cache management for cloud computing. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies (FAST '16)*, pages 355–369, 2016.

[3] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[4] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *Proceedings of the 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI '18)*, pages 389–403, 2018.

[5] Bryan T Bennett and Vincent J. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, 1975.

[6] Erik Berg and Erik Hagersten. Fast data-locality profiling of native execution. In *Proceedings of the 2005 ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '05)*, pages 169–180, 2005.

[7] Daniel Byrne. A survey of miss-ratio curve construction techniques. *arXiv preprint arXiv:1804.01972*, 2018.

[8] Dhruba Chandra, Fei Guo, Seongbeom Kim, and Yan Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture (HPCA '05)*, pages 340–351. IEEE, 2005.

[9] Peter J Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[10] Peter J Denning and Donald R Slutz. Generalized working sets for segment reference strings. *Communications of the ACM*, 21(9):750–759, 1978.

[11] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM footprint with NVM in Facebook. In *Proceedings of the Thirteenth European Conference on Computer Systems (EuroSys '18)*, pages 1–13, 2018.

[12] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI '11)*, volume 11, pages 24–24, 2011.

[13] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *Proceedings of the USENIX Annual Technical Conference (ATC '15)*, pages 57–69, 2015.

[14] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. Kinetic modeling of data eviction in cache. In *Proceedings of the USENIX Annual Technical Conference (ATC '16)*, pages 351–364, 2016.

[15] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*, pages 167–181, 2013.

[16] Qi Huang, Helga Gudmundsdottir, Ymir Vigfusson, Daniel A Freedman, Ken Birman, and Robbert van Renesse. Characterizing load imbalance in real-world networked caches. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks (HotNets '14)*, pages 1–7, 2014.

[17] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS Performance Evaluation Review*, 30(1):31–42, 2002.

[18] Ke Zhou, Yu Zhang, et al. LEA: A lazy eviction algorithm for SSD cache in cloud block storage. In *Proceedings of the IEEE 36th International Conference on Computer Design (ICCD '18)*, pages 569–572, 2018.

[19] Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '10)*, pages 65–76, 2010.

[20] Zaoxing Liu, Zhihao Bai, Zhenming Liu, Xiaozhou Li, Changhoon Kim, Vladimir Braverman, Xin Jin, and Ion Stoica. Distcache: Provable load balancing for large-scale storage systems with distributed caching. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies (FAST '19)*, pages 143–157, 2019.

[21] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM Computer Communication Review*, 45(3):52–66, 2015.

[22] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[23] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST '03)*, volume 3, pages 115–130, 2003.

[24] Frank Olken. Efficient methods for calculating the success function of fixed space replacement policies. 1981.

[25] Raj Parihar, Jacob Brock, Chen Ding, and Michael C Huang. Protection and utilization in shared cache through rationing. In *Proceedings of the 23rd International Conference on Parallel Architecture and Compilation Techniques (PACT '14)*, pages 487–488, 2014.

[26] D Shasha and T Johnson. 2Q: A low overhead high performance buffer management replacement algoritm. In *Proceedings of the Twentieth International Conference on Very Large Databases (VLDB '94)*, pages 439–450, 1994.

[27] SNIA. IOTTA. `http://iotta.snia.org/`.

[28] G Edward Suh, Srinivas Devadas, and Larry Rudolph. Analytical cache models with applications to cache partitioning. In *Proceedings of the ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 323–334, 2001.

[29] David K Tam, Reza Azimi, Livio B Soares, and Michael Stumm. RapidMRC: approximating l2 miss rate curves on commodity systems for online optimizations. *ACM Sigplan Notices*, 44(3):121–132, 2009.

[30] Tencent. CBS. `https://intl.cloud.tencent.com/product/cbs`.

[31] Elvira Teran, Zhe Wang, and Daniel A Jiménez. Perceptron learning for reuse prediction. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO '16)*, pages 1–12. IEEE, 2016.

[32] Michail-Antisthenis I Tsompanas, Christoforos Kachris, and Georgios Ch Sirakoulis. Modeling cache memory utilization on multicore using common pool resource game on cellular automata. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 26(3):1–22, 2016.

[33] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *Proceedings of the USENIX Annual Technical Conference (ATC '17)*, pages 487–498, 2017.

[34] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, pages 95–110, 2015.

[35] Wikipedia. Jhash. `https://en.wikipedia.org/wiki/Jenkins_hash_function`.

[36] Wolfram Mathworld. Weibull Distribution. `https://mathworld.wolfram.com/`.

[37] Xiaoya Xiang, Bin Bao, Tongxin Bai, Chen Ding, and Trishul Chilimbi. All-window profiling and composable models of cache sharing. *ACM SIGPLAN Notices*, 46(8):91–102, 2011.

[38] Xiaoya Xiang, Bin Bao, Chen Ding, and Yaoqing Gao. Linear-time modeling of program working set in shared cache. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*, pages 350–360. IEEE, 2011.

[39] Xiaoya Xiang, Chen Ding, Hao Luo, and Bin Bao. Hotl: a higher order theory of locality. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pages 343–356, 2013.

[40] Juncheng Yang, Reza Karimi, Trausti Sæmundsson, Avani Wildani, and Ymir Vigfusson. Mithril: mining sporadic associations for cache prefetching. In *Proceedings of the Symposium on Cloud Computing (SOCC '17)*, pages 66–79, 2017.

[41] Chencheng Ye, Jacob Brock, Chen Ding, and Hai Jin. Rochester elastic cache utility (recu): Unequal cache sharing is good economics. *International Journal of Parallel Programming*, 45(1):30–44, 2017.

[42] Seyed Majid Zahedi and Benjamin C Lee. REF: Resource elasticity fairness with sharing incentives for multiprocessors. *ACM SIGPLAN Notices*, 49(4):145–160, 2014.

[43] Yutao Zhong, Xipeng Shen, and Chen Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):1–39, 2009.

[44] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. Demystifying cache policies for photo stores at scale: A Tencent case study. In *Proceedings of the International Conference on Supercomputing (ICS '18)*, pages 284–294, 2018.

[45] Yuanyuan Zhou, James Philbin, and Kai Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the USENIX Annual Technical Conference, General Track*, pages 91–104, 2001.

[46] Sergey Zhuravlev, Sergey Blagodurov, and Alexandra Fedorova. Addressing shared resource contention in multicore processors via scheduling. *ACM Sigplan Notices*, 45(3):129–142, 2010.