SACache: Size-Aware Load Balancing for Large-Scale Storage Systems

Yihong Su[1], Hang Jin[1], Fang Liu[1](✉), and Weijun Li[2]

[1] School of Computer Science and Engineering, Sun Yat-Sen University, Guangzhou, China
liufang25@mail.sysu.edu.cn

[2] Shenzhen Dapu Microelectronic Co., Ltd., Shenzhen, China

Abstract. The fast cache could be used in storage clusters to alleviate load imbalance caused by highly-skewed requests between storage nodes. In a smaller cluster, we can use a single cache node to solve the I/O bottleneck caused by load imbalance. However, in a Large-scale cluster, we may need more than one cache node to afford enough capacity, which brings a new load balance problem in cache nodes. DistCache successfully solved this problem by applying the power-of-two-choices. In the above storage clusters, cache nodes cache the hottest objects while ignoring the size of objects, which leads to poor performance when meeting objects with variable sizes. We present SACache, a size-aware mechanism for large-scale storage clusters, which can improve I/O performance by maximizing the benefit of the unit cache. In this mechanism, we set an object admission filter to filter out objects with lower caching benefit. To adapt to changing request patterns, we record recently requested objects and their size, then replay those requests periodically in a cache simulator to find the best cache admission parameter using a greedy algorithm and apply it to the object admission filter. We apply this mechanism in a prototype distributed storage system. Experimental results show that it can increase the system's overall bandwidth when the object's size is different.

Keywords: Load balancing · Size-aware · Large-scale storage · Caching

1 Introduction

Distributed object storage systems provide scalable storage for modern data-intensive applications where data is stored as key-value pairs, and applications can access values from a storage cluster that spans thousands of nodes with a unique key [1–4]. Key-value storages are already widely deployed in social networking, data analytics, Web search, download centers, and other applications. In the real world, the request loads are high-skewed, few popular objects may receive most of the requests, which brought challenges to the load balancing. However, skewed access leads to a spatial locality. Thus, we can load hot objects into the cache, improve the system bandwidth, and reduce latency, resulting in a better user experience and meet stricter service-level objectives (SLOs).

In a typical object storage system, HDD is used as the storage medium with a large capacity with low cost; however, it has a high access latency. Due to disk seek time (about 10 ms), HDD is not friendly to random access. Besides, the access speed of HDD is slower than RAM. If we cache some objects into RAM, we can reduce disk I/O's performance overhead. Meanwhile, caching hot objects can also be used as a load balancing strategy for a storage cluster. For a cluster with n storage nodes, caching $O(n\log n)$ hottest objects can balance the system, regardless of the number of objects [5].

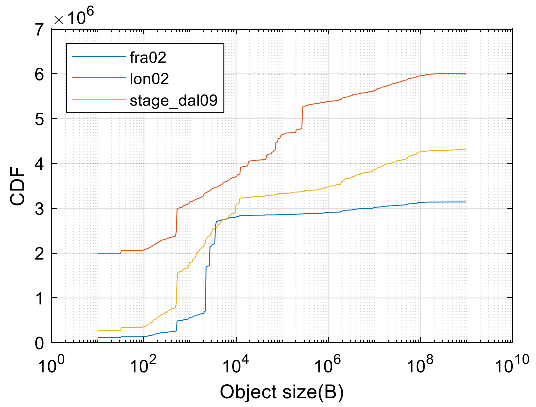

Fig. 1. Cumulative distribution for object sizes in three different data centers, traces are from an IBM Docker Registry.

In an object storage system, cache replacement algorithms (e.g., LRU and LFU) prioritize hot objects and evict objects in the cache. In real-world applications, there are also differences in size between objects besides differences in popularity. Figure 1 shows the object distribution of three traces from the IBM Docker Registry production [6]. We can find that the range of object size is vast, from KB magnitude to GB magnitude. In this case, if using the unmodified cache management algorithm, putting a large object into the cache can cause multiple small objects to be evicted. If we need to access these small objects once again, we need to access HDD storage nodes, resulting in a large number of HDD random access, which will decrease the system's overall bandwidth.

Because the object size range in an object storage system is wide, to maximize the value of the small but expensive RAM, we need a mechanism to evaluate the benefit of an object being cached and decide whether to be admitted by the cache. At the same time, this mechanism should be able to automatically adapt to different data load characteristics and dynamically adapt to the change of load characteristics over time. Therefore, it is necessary to analyze the benefits of storage systems based on HDD and RAM characteristics and design a cache admit mechanism to maximize the system performance with limited cache capacity.

We present SACache, an easy-to-understand cache management strategy suitable for storage systems with large size differences. Based on a lightweight cache node simulator (shadow cache), SACache searches for the current optimal caching parameters by replaying the recent trace of object request records (including object name and size) in the cache node. It can be used as both a cache management mechanism for stand-alone object storage systems and an optimization method for large distributed object storage.

SACache adds a cache admission mechanism based on the cache replacement algorithm, ensuring that small objects have a greater probability of being cached. Simultaneously, there is a caching parameter in the admission mechanism, which determines how much object size affects the probability of being admitted. The shadow cache is the key to finding the appropriate caching parameters, and there is a shadow cache in each cache node to simulate the node's performance under a specific caching parameter. When searching for caching parameters, we compare the shadow cache's performance under different caching parameters to select the optimal caching parameter. Fine-grained search interval can guarantee higher precision but may bring unacceptable time overhead. Based on empirical observation, we implement a fast search algorithm that achieves high precision through fewer search times so that the parameter search time is completely acceptable.

1. We implement an object storage prototype system integrating SACache, measure performance improvement and stability of SACache in the simulation environment (for both single-node and storage cluster). Our contribution can be summarized as follows:
2. We analyze and summarize a benefit formula of the object storage system with a caching mechanism.
3. We designed an efficient shadow cache to simulate the behavior of real cache nodes.
4. We implement a fast parameter search algorithm for searching the optimal caching parameters.
5. We evaluated the performance improvement of SACache under different loads in a simulation environment for both single-node and storage clusters.

2 Background and Motivation

2.1 Cache Based Load Balancing

Partitioning is frequently-used in the extensible object storage system, where each node independently provides the storage and query of partial objects. Uneven request loads can lead to performance differences between nodes, resulting in reduced cluster throughput and increased latency. A hash-based partitioning function is sufficient to partition the objects evenly in a cluster with constant storage nodes. For applications that require scalability, consistent hashing is used to ensure that objects can be partitioned evenly after adding or removing a server node. However, these schemes are insufficient for the load imbalance caused by the difference in popularity among objects.

Popularity differences between objects are usually highly skewed in real-world internet applications. Object access Popularity tends to follow Zipf distribution, and a small amount of hot object will be requested frequently. The skewed access makes small capacity high-speed caching system accepts many requests, which avoids many requests sent directly to the back-end storage nodes by taking full advantage of the cache characters of low latency and high bandwidth. Some researchers have used a small but fast enough cache node in a cluster as a front-end load balancer, and the lower bound of cache size depends only on the number of back-end storage nodes in the system but independent of the total number of objects in the cluster. Caching at least $O(n\log n)$ hottest objects can ensure load balancing among n storage nodes [5]. Based on this theory, SwitchKV [7] uses RAM cache to balance the SSD storage nodes, and NetCache [8] uses a faster in-switch cache to balance storage nodes based on RAM. Furthermore, DistCache [9] uses two independent hash functions to partition objects in two equivalent cache clusters.

2.2 Cache Systems with Variable Object Sizes

Existing caching algorithms mostly focus on eviction policies (i.e., LRU, LRU-K [10], SLRU [11], ARC [12]), which tend to have no admission policy and cache all accessed objects. At the same time, these studies are designed for caching similar object sizes. In recent years, researches on caching with variable object sizes have emerged. The size-based AdaptSize [13] caches an object with a probability of $\mathrm{e}^{\mathrm{size}/\mathrm{c}}$, and adjusts the parameter c based on a Markov model. Rl-Cache [14] implements an admission policy based on reinforcement learning, and it uses more features to characterize objects in a request trace. However, the optimization goals of the above AdaptSize, RL-Cache are the object hit ratio, which is not consistent with load balancing.

In order to improve the performance of the load balancer when the object size is different, we add an admission filter to the front-end cache. The existing related work about cache admission policy is primarily for the CDN scenario and aims to maximize the object hit ratio. Unlike them, SACache is designed for optimizing load-balancing performance for storage systems with objects of different sizes.

2.3 Motivation

In cache systems, there are at least two kinds of storage media with different characteristics: one is a high speed, low latency but expensive medium, which is suitable for cache; the other is a medium with low speed, high latency but low price, it is suitable for storage. The above works do not consider the size of the object, however, in some Internet applications, for example, in a storage cluster of web resources, the size of objects (web pages, pictures, videos) varies greatly. Meanwhile, the storage medium's access characters determine that when objects have the same popularity, caching many small objects incurs a more significant overhead than caching one large object. For example, caching ten small objects of size one does not yield the same benefits as caching one big object of size ten, this is mainly due to the low latency nature of the cache media, which we will explain it in the following paragraphs (Table 1).

Table 1. Notation used for the analysis.

Symbol	Meaning
t_s	Latency of storage medium
t_c	Latency of cache medium
k_s	Inverse of the storage medium bandwidth
k_c	Inverse of the cache medium bandwidth
s	Object's size
v	Object's popularity
e	Benefit of unit cache capacity

Suppose that in a storage-cache system with a cache node and a storage node, HDD is used as the storage medium, and DRAM is used as the cache medium. Based on this system, we ignore the network overhead to analyze the benefits of caching an object. For HDD, each data Access Time can be divided into three parts: seek time (T_s), rotational time ($1/(2r)$), and transfer time ($B/(rN)$), where r is the rotation speed of the disk, N is the number of bytes on a track, and B is the number of bytes form an IO transfer. Seek time and rotational time determine the average access latency, while transfer time determines the transmission bandwidth. Typical access latency is about 10 ms, and the transfer bandwidth is about 200 MB/s. The time cost of an s bytes IO transfer can be formulated as $\mathrm{t_s} + \mathrm{k_s} * \mathrm{s}$. For RAM with random access character, the time cost can also be formulated $\mathrm{t_c} + \mathrm{k_c} * \mathrm{s}$, The current common two-channel DDR4 has about 50 GB/s bandwidth and 100 ns latency, so $\mathrm{t_s} \gg \mathrm{t_c}$ and $\mathrm{k_s} > \mathrm{k_c}$.

In the above object caching system with limited memory, it is important to fully utilize cache capacity to reduce the overall request wait time. We want to define a metric for objects in cache to evaluate the "cost performance." The *benefit* of caching an object comes from the wait time saved by requesting the object directly from the cache; if the object is requested v times over a period of time, the saved time is $\mathrm{v} * (\mathrm{t_s} - \mathrm{t_c} + \mathrm{k_s} * \mathrm{s} - \mathrm{k_c} * \mathrm{s})$; and the *cost* is s, the capacity occupied by the object. So we can define a $\frac{\mathrm{benefit}}{\mathrm{cost}}$ ratio (i.e., the benefit of unit cache) e for objects in the cache, and it is the ratio of the IO time divided by used space of caching an object, we have:

$$e = \frac{v * (t_s - t_c + k_s * s - k_c * s)}{s} \tag{1}$$

For a specific object storage system, t_s, t_c, k_s, and k_c are measurable constants. It is easy to see that when v is constant, e decreases as s increases; which indicate that for two objects with the same popularity, the smaller object has a higher $\frac{benefit}{cost}$ ratio.

To prevent low-value objects from occupying cache space, we want the e value of the objects kept in the cache to be higher than a lower bound $\underline{e}$. To find out the condition for different objects to reach this lower limit, we need to study the popularity condition for different objects of different sizes to have the same value of e. We describe popularity v as a function of object size s.

$$v(s) = \frac{e * s}{(t_s - t_c) + (k_s - k_c) * s} \tag{2}$$

If the latency and bandwidth of the storage system and the cache system is known, we can give an object of size s the popularity v needed to keep the unit cache benefit equal to$\underline{e}$. Figure 2 shows an equal e curve of a storage-cache system, where the X-axis is the size of the object, the Y-axis is the relative popularity, and the benefit is consistent at any point in the curve. And the background color on Fig. 2 corresponds to the unit cache benefit e under a coordinate $\langle s, v\rangle$, darker color indicate higher$e$.

We can design a cache admission policy based on the $v(s)$ function, the keynote is that only a object of size s reaches the popularity v, which makes benefit e higher than $\underline{e}$, the object can be admitted by cache. The key is to find an appropriate $\underline{e}$ that maximizes the storage node bandwidth under the current load characteristics (e.g., object size distribution and request skew). Too small $\underline{e}$ will disable the cache admission policy and too big $\underline{e}$ will make it almost impossible for an object to be cached. In Sect. 3, we will show the specific design of the cache admission mechanism which we call SACache.

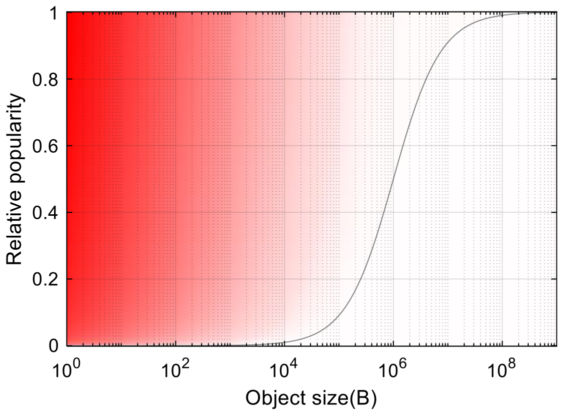

Fig. 2. An equal benefit curve, reflecting the popularity to ensure the same unit cache benefit as the object size increases. And the depth of the red color reflects the level of the unit cache benefit e. (Color figure online)

3 SACache System Design

The SACache mechanism is applied in all cache nodes, and the fundamental idea of this mechanism is to decrease the admission probability of big objects. In applications with different sizes of objects, accepting a large object when the cache space is insufficient will lead to the eviction of many small objects, which may be requested again soon. The cache admission policy is designed in cooperation with the load balancing mechanism to maximize the system's load balancing. In this section, we provide an overview of SACache's architectural design and implementation, as well as two use cases for SACache.

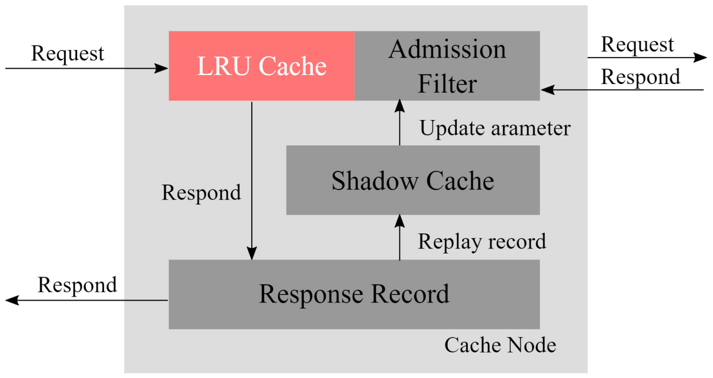

Fig. 3. Cache node architecture.

3.1 SACache Architecture

Figure 3 shows an overview of the SACache node's architecture. SACache adds an admission policy to LRU, called the admission filter, which intercepts PUT operations to LRU. The admission filter has a cache parameter c, which, together with the object's *size*, determines the request frequency for the object to be admitted. Finding the appropriate parameter c is the key to maximize load balancing performance. Cache node will record recent requests, replay them in the shadow cache, and use a greedy algorithm to search the best cache admission parameter to maximize the optimization object (i.e., bandwidth).

Object Admission Filter. In SACache, small size objects will have more chances to be admitted by the cache node. Based on the Eq. (2) in Sect. 2, we design an admission function. For an object of size *size* and popularity K, it will be admitted by the cache if $K > v(size)/v(c)$. The cache parameter c ranges between 1 and *cache_size*, and it is essential to find optimal parameter c_{opt}; we use an accelerated search method to search for c_{opt} in the logarithmic space with shadow cache, which we will describe later.

Another important thing is to find a method to sense the popularity of an object. To achieve this, we use a recent visit counter to provide a popularity reference for cache admission; The counter is essentially a FIFO queue of length N, and it will record at most N unique latest cache miss objects' visit count. For cache miss in the cache node, the visit counter will be refreshed after fetching the object from the storage node, then the visit count of the uncached object increased by one. If the uncached object does not exist in the FIFO queue, it will be pushed a new record to the queue; and of course, if the FIFO queue's length has reached N after the push, the last record entry will be deleted before push operation. The recent visit counter is equivalent to the popularity estimation of an object. An object in a recent visit counter will be decided to be admitted after it is refreshed. Specifically, if an object's visit count is greater than $v(size)/v(c)$, it will be admitted by the cache, and then the record will be removed from the recent visit counter. In this way, the popularity needed to put an object into the cache increases linearly with the object's size, and there is a static optimal parameter c that maximizes the optimization target for a static object distribution and request distribution.

Shadow Cache. To find the optimal parameter c, we record the recent request traces, and each item includes the object name and object size. To save storage space, we map the object name to a 64-bit id and then storage the object size as a 64-bit unsigned integer, so the record item is a 16-byte tuple $\langle id, size \rangle$. When we have enough request records, we can choose a parameter c and replay the traces in a shadow cache. By simulating a request in the shadow cache, we can know whether the requested object is cached or not. Then we can know the system's overall bandwidth under a specific caching parameter c in these traces. Experiments show that the shadow cache's performance overhead is not too large, and the simulation of 250k requests can be completed within 40 ms.

Greedy Search. In SACache, the caching parameter c is in the range of $[1, cache\_size]$. We take log-scale caching parameter $\log(c)$ as the x-axis and the optimization goal (e.g., hit ratio or bandwidth) as the y-axis. In order to get high precision results quickly, we designed a greedy search algorithm. The algorithm consists of multiple iterations: in each iteration, c is selected at equal log-scale intervals; the optimization goal under parameter c is obtained through the shadow cache, the c_{opt} that maximizes the optimization goal is the optimal parameter in this iteration. In the next iteration, the search range will change; the new search range is centered around $\log\left(c_{opt}\right)$, and the range will be halved. In this way, each iteration's execution time is similar, but each iteration's accuracy is doubled than the previous iteration. We use the above search algorithm in our system, and we search 16 parameters in each iteration and have five iterations. So we run the simulator 50 times, but the search accuracy is equivalent to 256 times ($256 = 16 \times 2^{5-1}$). The pseudocode is shown in Appendix A.

3.2 Implementation

Cache Node. The cache node caches objects as key-value pairs (e.g., A: value and B: value). With a size-aware admission policy, small objects will be cached preferentially. To manage the cached objects in memory, we use a traditional LRU algorithm. Specifically, we implement an LRU cache using two containers (list and unordered_map) in C++ STL. The list in unordered_map is a doubly-linked list that is efficient at inserting and deleting when the address of the element is known. We use unordered_map to map the object name to the object address, so we can locate the object in constant time complexity. On a cache miss, the cache node visits the storage node and relays the object directly to the client, rather than having another access storage node on the client to request the object. At the same time, the object admission filter will decide whether to save the missed object. We use a single-consumer task queue to operate the cache, which ensures thread safety. The cache node will record information such as object requests, the number of cache hits, and the most recent hit ratio to measure SACache's performance.

Client. The client is implemented to test our storage system by requesting objects from the cache node through the HTTP API. The client can either replay the existing trace to generate requests or generate trace by configuring the object size distribution and access popularity distribution. Besides, we implement DistCache by storing the cache node's address on the client.

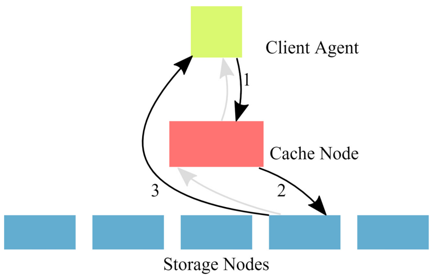

Fig. 4. Cache miss and object can't be cached.

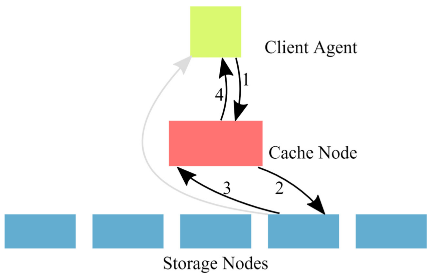

Fig. 5. Cache miss and object can be cached.

3.3 Request Path

In SACache, we used a client agent to connect to numerous clients and the object storage cluster. In order to reduce the overhead caused by frequent network interactions within the system, long-lived connections are established between the agent and the cache nodes and storage nodes in the cluster. On a cache hit, the cache node returns the object kept in memory to the agent. For a cache miss, the cache node will update its visit counter and forwards the request to the storage node where the accessed object exists.

Due to the existence of the admission filter mechanism in SACache, an object returned directly to the cache node by the storage node in case of a cache miss could be refused by the cache admission filter, which causes additional transfer overhead. We want the storage node to have the ability to determine whether the current object needs to be cached or not. If it is not cached, returning the object directly to the client agent can save an object transfer overhead. In the prototype system implementation, the cache node will append the parameter c and the object's visit count K (from visit counter) to the request to the storage node. In this way, the storage node can judge the next network transmission direction by comparing the object access counts K and $v(size)/v(c)$. When $K < v(size)/v(c)$, the object can be directly returned to the client agent; Otherwise, the object needs to be returned to the cache node, which forwards the object to the client agent. Figure 4 and Fig. 5 show two types of routing processing for a cache miss; the pseudocode is shown in Appendix B.

3.4 Use Cases

SACache is a general mechanism for increasing system bandwidth in scenarios with varying object sizes. It can be used in both single cache nodes and distributed cache nodes, which we describe next.

Single Cache Node. Compared with storage nodes, storage media of cache nodes perform better in bandwidth and latency, which can be used to ensure load balancing between storage nodes by caching hot objects into cache nodes. For applications with a small number of objects and a large request heat tilt, a single high-speed cache node may be used to load $O(n\log n)$ of the hottest objects to balance the request pressure between storage nodes. A single cache node does not avoid load balancing between cache nodes and is easier to handle on cache consistency, but it cannot provide sufficient bandwidth and reliability for a larger cluster.

Distributed Cache Nodes. In large-scale distributed storage, a single cache node is not enough to provide sufficient cache capacity. The bandwidth also needs to be higher than the cumulative bandwidth of all storage nodes as the cache layer. Therefore, multiple cache nodes are needed to realize load balancing among storage nodes. Simply copying hot objects to all cache nodes does not take full advantage of expensive memory and incurs write time synchronization overhead; and if objects are divided into different cache nodes by a hash function, load imbalance will occur between cache nodes under a highly skewed load. We can refer to DistCache to implement a distributed cache system. Because there is no coupling between the cache nodes, SACache can be easily extended to a distributed cache without additional modifications.

4 Evaluation

4.1 Experimental Setup

Our testbed is a server machine equipped with an 8-core 16-thread CPU (AMD 3700x), 16 GB memory (Samsung 16 GB DDR4-3200 memory), and we use G++ 9.3.0 compiler and running the simulation program on ubuntu20.04. We conduct four experiments to show the characteristics of SACache in different aspects. In the following subsections, we explore the performance benefits of SACache and the costs of introducing the SACache mechanism. The experimental results show that our SACache can improve the load balancing performance within the accepted performance cost.

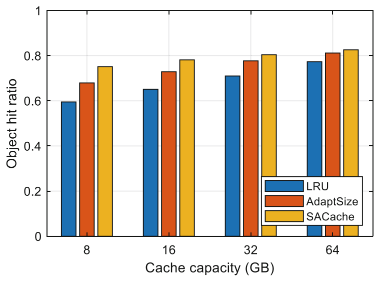

Fig. 6. Object hit ratio for single cache node.

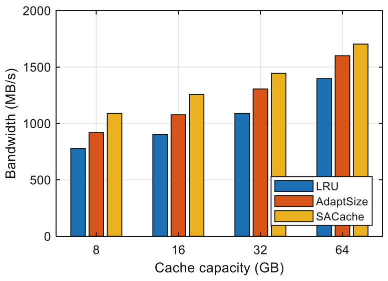

Fig. 7. Bandwidth for single cache node.

4.2 Experimental Result

Performance on Single Cache Node. In this section, we evaluate the performance (including hit ratio and throughput) of SACache compared to AdaptSize and LRU cache on an object storage system with a single cache node and ten storage nodes. We generate random request trace with Zipf distribution; under this workload, a small number of objects are accessed frequently, and most objects are rarely accessed. Simultaneously, small objects account for more bytes and large objects for less, consistent with the trace from the IBM Docker registry. Traditionally, the hit ratio is an important indicator to evaluate cache replacement algorithms. In a cache system for objects of the same size, a higher hit ratio means a higher IOPS performance. Figure 6 shows the hit ratio performance of three different caching policies, with cache sizes ranging from 8 GB to 64 GB. We use a log-scale x-axis to present a wide range of cache capacities. We can observe from Fig. 6 that for the three different caching policies, as the cache's capacity increases, the cache can keep more objects, and less cache replacement occurs, which leads to a higher hit ratio. Simultaneously, with the increase of cache capacity, the hit rate will reach saturation and approach an upper bound. Considering the object size difference, both AdaptSize and SACache can achieve a higher hit ratio than LRU cache for different cache capacity. SACache improves the mean hit ratio and max hit ratio by 16.7%

and 26.2% over LRU cache, respectively. Moreover, compared to AdaptSize, SACache improves the mean hit ratio by 10.6% and improves the max hit ratio by 5.8%. However, in applications with different object sizes, the hit ratio is insufficient to indicate performance, so we also record the cache system's bandwidth with different policies. Figure 7 shows the bandwidth of the system under different cache capacities. As shown in the hit ratio experiment, SACache is superior to AdaptSize and LRU cache in bandwidth performance. Compared to LRU, SACache's bandwidth is 33.5% higher on average and 39.9% higher at most.

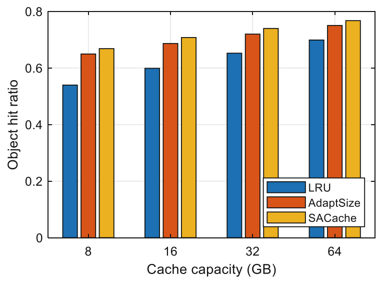

Fig. 8. Object hit ratio for multi-node cache.

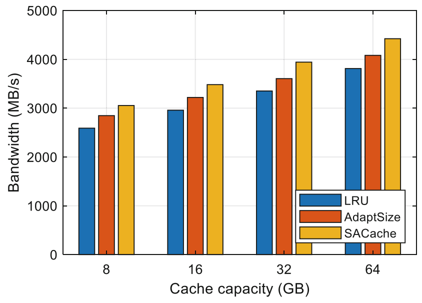

Fig. 9. Bandwidth for multi-node cache.

Performance on Distributed Cache Nodes. To verify the applicability of SACache in large-scale storage systems, we examine the performance of SACache for distributed cache nodes. The request mode of the generated trace is the same as that on a single cache node, but the scale of the dataset is larger, with both the number of unique objects and the length of the trace greater than the set in the single-node test. The cache nodes in SACache are not independent of each other, and there is no data interaction, which allows the multi-node cache to be extended without modifying the nodes. Referring to DistCache's design, we realize a load-balancing distributed cache system, which contains a total of 16 cache nodes in two layers. The client agent records the load of the cache node and chooses the node with a lower load adaptively according to power-of two-choices [15] when requesting objects. Figure 8 and Fig. 9 respectively show the bandwidth and hit ratio performance for distributed caches. Similar to the performance on a single node, SACache is superior to AdaptSize, suggesting that SACache can also be used as a new optimization approach in distributed cache systems.

Bandwidth Scalability of Different Policies. Figure 10 shows the system bandwidth scalability using cache as the load balancer. Different admission policies show good scalability under Zipf access popularity. This is because the front-end cache node absorbs most of the popular objects, making the remaining cold objects distributed evenly on the back-end storage node. However, SACache still performs better in scalability than the other two policies.

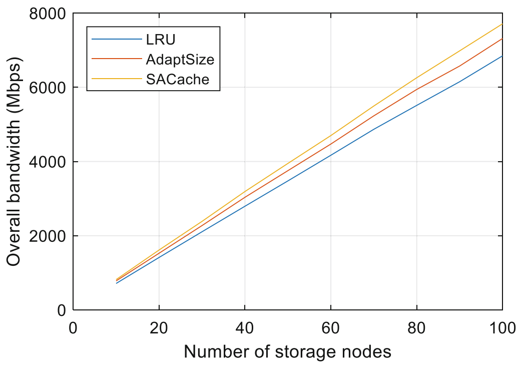

Fig. 10. Bandwidth for single cache node.

Parameter Updating Time Cost. Figure 11 and Fig. 12 show the cache admission parameter updating time cost of SACache, with parameter updated every specific number of requests to track changes in the load. In our prototype system tests, we have 100K requests per minute. Compared to a simple LRU-based cache system without object admission filter, SACache inevitably incurs additional time cost to handle requests and search for the best caching parameters. However, the added cost of handing requests itself is negligible, so we focus on the optimal parameter search time cost. We use the trace collected to generate requests to evaluate the elapsed time of the SACache parameter search. For each parameter search, the cache node will replay the last 250k requests (in about 150 s) recorded and calculate the hit rate; we use a greedy algorithm to speed up finding the best caching parameter *c* with the highest hit rate. The experimental result shows that 99.3% of 30369 parameter search tests based on different traces can

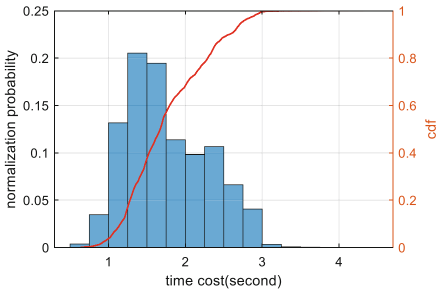

Fig. 11. The time cost of update the cache admission parameter every 150 s. We repeat the experiment and showed the distribution and cumulative probability sum of time consumption in the figure.

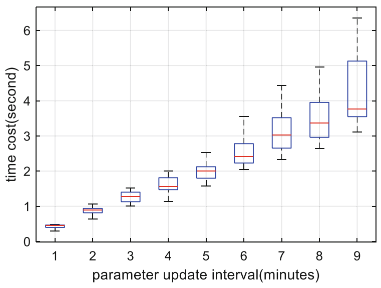

Fig. 12. As the interval between update parameters increases, the time cost of the parameter search increases linearly because longer records are used for parameter searches.

be completed within 3 s, and the maximum search time is 4.31 s. It will take several minutes for the cache node to receive 250k requests in a real system, and the parameter search could be executed in another process, so the parameter searching in SACache would not be a performance bottleneck.

5 Related Work

Load balancing is crucial for distributed storage systems. Previous work indicates that in a system owning n storage nodes, caching $O(n\log n)$ hottest objects can avoid load unbalancing. For larger distributed storage systems, multiple cache nodes are needed to provide sufficient capacity for the storage cluster. To achieve load balancing among cache nodes, DistCache proposes a two-layer caching architecture and uses independent hash functions for request routing. However, in object storage systems, the size of objects is often different. For storage systems with variable object sizes, the size-aware load balancing mechanisms can achieve higher performance or lower cost. AdaptSize is the first size-aware cache admission policy for hot object caching in CDN. To achieve a higher object hit ratio, it searches for the optimal caching parameters based on a novel Markov cache model and continuously adjusts the caching parameters according to the change of request patterns. FOO and PFOO [16] gave the theoretical upper bound hit ratio of caching with variable object sizes and reveal that the current caching system is still far from optimal. To achieve cost-friendly load balancing by caching hot objects, we design SACache with a focus on object size differences; the experimental results show that the performance of SACache is better than AdatpSize.

6 Conclusion

Modern internet applications are becoming more data-intensive, and in-memory caching is crucial for improving applications' storage performance. We present SACache, an efficient caching mechanism that considers the difference in object size and is aimed at maximizing system bandwidth. SACache takes advantage of a simple admission filter that evaluates the benefits of caching an object when it is admitted. We present the design and implementation of SACache and evaluate the load-balancing performance improvement of SACache in large scale distributed cache through experiments.

Acknowledgement. This work is supported by The Key-Area Research and Development Program of Guang Dong Province (2019B010107001), National Key Research and Development Program of China (2019YFB1804502), and National Natural Science Foundation of China (61832020, 61702569).

Appendix A

Algorithm 1 Search for c_{opt} (optimal parameter).

```
Input: The recent request trace
Output: The optimal cache admission parameter
 1: function SearchOptimalParameter($R$)
 2:     $N \leftarrow$ number of iterations
 3:     $M \leftarrow$ number of searches per iteration
 4:     $c_{opt} \leftarrow 0$
 5:     $c_{min} \leftarrow 0$
 6:     $c_{max} \leftarrow cache\_size$
 7:     for $it = 1$ to $N$ do
 8:         for $c$ in LogSpace($c_{min}, c_{max}, M$) do
 9:             if SimScore($R, c$) > SimScore($R, c_{opt}$) then
10:                 $c_{opt} \leftarrow c$
11:             end if
12:             $c_{min} \leftarrow$ MAX($0, c_{opt} - cache\_size/2^{it+1}$)
13:             $c_{max} \leftarrow$ MIN($cache\_size, c_{opt} + cache\_size/2^{it+1}$)
14:         end for
15:     end for
16:     return $c_{opt}$
17: end function
18:
19:
20: function SimScore($R, c$)
21:     $s \leftarrow 0$                                  ▷sum of object size
22:     $t \leftarrow 0$                                  ▷sum of access time
23:     ShadowCache.SetParameter($c$)
24:     for $i = 1$ to $N$ do
25:         $s \leftarrow s + R[i].size$
26:         if ShadowCache.GET($R[i].key, R[i].size$) then
27:             $t \leftarrow t +$ CacheAcessTime($R[i].size$)
28:         else
29:             $t \leftarrow t +$ StorageAcessTime($R[i].size$)
30:         end if
31:     end for
32:     return $t/s$
33: end function
```

Appendix B

Algorithm 2 Path choosing for storage nodes.

```
1: $K \leftarrow$ visit count of $object$
2: if $K < v(size)/v(c)$ then
3:     SendObjectTo(object, clientAgent)
4: else
5:     SendObjectTo(object, cacheAgent)
6: end if
```

References

1. Ghemawat, S., Gobioff, H., Leung, S.-T.: The google file system. In: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles, pp. 29–43 (2003)
2. Hastorun, D., et al.: Dynamo: Amazon's highly available key-value store. In: Proceedings of SOSP. Citeseer (2007)
3. Beaver, D., Kumar, S., Li, H.C., Sobel, J., Vajgel, P., et al.: Finding a needle in haystack: Facebook's photo storage. In: OSDI, vol. 10, pp. 1–8 (2010)
4. Factor, M., Meth, K., Naor, D., Rodeh, O., Satran, J.: Object storage: the future building block for storage systems. In: 2005 IEEE International Symposium on Mass Storage Systems and Technology, pp. 119–123 (2005)
5. Fan, B., Lim, H., Andersen, D.G., Kaminsky, M.: Small cache, big effect: provable load balancing for randomly partitioned cluster services. In: Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC 2011. Association for Computing Machinery, New York (2011)
6. Anwar, A., et al.: Improving docker registry design based on production workload analysis. In: 16th USENIX Conference on File and Storage Technologies (FAST 2018), Oakland, CA, pp. 265–278. USENIX Association, February 2018
7. Li, X., Sethi, R., Kaminsky, M., Andersen, D.G., Freedman, M.J.: Be fast, cheap and in control with switchkv. In: 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2016), Santa Clara, CA, pp. 31–44, USENIX Association, March 2016
8. Jin, X., et al.: Netcache: balancing key-value stores with fast in-network caching. In: Proceedings of the 26th Symposium on Operating Systems Principles, pp. 121–136 (2017)
9. Liu, Z., et al.: Distcache: provable load balancing for large-scale storage systems with distributed caching. In: 17th USENIX Conference on File and Storage Technologies (FAST 2019), Boston, MA, pp. 143–157. USENIX Association, February 2019
10. O'neil, E.J., O'Neil, P.E., Weikum, G.: An optimality proof of the LRU-K page replacement algorithm. J. ACM (JACM) **46**(1), 92–112 (1999)
11. Morales, K., Lee, B.K.: Fixed segmented LRU cache replacement scheme with selective caching. In: 2012 IEEE 31st International Performance Computing and Communications Conference (IPCCC), pp. 199–200 (2012)
12. Megiddo, N., Modha, D.S.: ARC: a self-tuning, low overhead replacement cache. Fast **3**, 115–130 (2003)
13. Berger, D.S., Sitaraman, R.K., Harchol-Balter, M.: Adaptsize: orchestrating the hot object memory cache in a content delivery network. In: 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2017), Boston, MA, pp. 483–498. USENIX Association, March 2017

14. Kirilin, V., Sundarrajan, A., Gorinsky, S., Sitaraman, R.K.: RL-cache: learning-based cache admission for content delivery. IEEE J. Sel. Areas Commun. **38**(10), 2372–2385 (2020)
15. Mitzenmacher, M.: The power of two choices in randomized load balancing. IEEE Trans. Parallel Distrib. Syst. **12**(10), 1094–1104 (2001)
16. Berger, D.S., Beckmann, N., Harchol-Balter, M.: Practical bounds on optimal caching with variable object sizes. Proc. ACM Meas. Anal. Comput. Syst. **2**(2), 1–38 (2018)