CircularCache: Scalable and Adaptive Cache Management for Massive Storage Systems

Liqiong Liu, Xiaoyang Qu, Yubiao Zhang, Xiaodong Yi, Siwang Zeng, Jiguang Wan*, Changsheng Xie
National Laboratory for Optoelectronics
Huazhong University of Science and Technology
Wuhan, Hubei, P.R. China.
Email:{liulqiong, quxiaoy}@gmail.com, {jgwan,cs_xie}@mail.hust.edu.cn

Abstract—**In order to enhance the performance of HDD-based storage systems, low-latency and high-IOPS SSDs are usually deployed as a cache above HDDs. With explosive data growth, a large-scale SSD-based cache tend to adopt partition management for overall cached data distribution across multiple cache nodes. We proposed an adaptive and scalable SSD-based cache called CircularCache, which distributes hot data across multiple cache nodes. The hotter virtual disks deserve more allocated free space in the SSD-cache. This paper exploited a dynamic replacement algorithm called VBQ(VDI-Based Queues) to manage the SSD-cache. The VBQ scheme manages the SSD-cache by dynamically manipulating the upper-bounds and lower-bounds of multiple queues based on the total access number of virtual disks. To mitigate negative impacts of destaging on overall storage performance, the dirty data in the cache will be written back to data nodes during idle time. At the same time, we utilize the redundant storage space in the data nodes as logging area to retain reliability of the dirty data on the SSD-cache. The prototype of CircularCache is implemented based on Sheepdog. Experimental results show that CircularCache offers a performance improvement by up to 270% compared with the standard distributed storage system without an SSD-based cache.**

I. INTRODUCTION

The Infrastructure-as-a-Service(IaaS) can offer virtualization platform interfaces. With IaaS, clients can share available hardware(physical) resources: storage resources, computing resources, and network infrastructures. The infrastructure resources are owned and managed by providers and can be purchased by customers on-demand. Several service corporations have provided this service, such as Amazon Web Services(AWS) and Microsoft Azure. With development of IaaS, block-level distributed storage systems become more and more important. For distributed storage systems, the block-level interface can be adapted to heterogeneous client applications.

For block-level distributed storage systems, virtual disks [22] are proposed as interfaces to access storage resources. For storage resource virtualization, virtual disks can make the client's view of storage transparent to physical storage resource. Consumers can purchase physical resource on demand, which make the resource allocation more flexible.

Solid State Drives(SSDs) have attracted large amount of attention due to its high performance and low energy consumption. Because of high price and low capacity of SSDs, it is popular to utilize high performance of SSDs and high capacity of HDDs to architect hybrid storage systems. SSD-based arrays have been proposed to construct high-performance storage system. However, SSDs are too expensive compared to HDDs so far, they cannot replace the HDDs in a few years. In addition, the amount of cold data accounts for a large fraction of total data. This may reduce cost-performance of the overall storage system. Thereby, SSDs are widely used for hybrid storage systems where the SSDs cache hot data. In this paper, we use several SSD nodes as a global cache above HDD nodes.

* Corresponding author

With the explosive growth in internet use, the amount of new data is generated exponentially over time. With the explosive data growth, conventional cache data partitions fail to adapt to rapid data growth. In other words, the cache size should be scalable and adaptive. For a large-scale cache consists of multiple nodes, it is challenging to efficiently distribute the data across multiple cache nodes. In addition, the associated replacement policy should consider not only the recency and frequency but also the characteristics of access patterns in virtual environments.

With sharply increasing amount of data, the global SSD-cache should be scalable and adaptive. We exploit a scalable SSD-based cache called CircularCache, which supports hot data distributions across multiple cache nodes. For CircularCache, it is challenging to distribute hot data evenly and to balance the workload, thereby this paper employs consistent hashing as the cache partition policy. In addition, this paper exploits a dynamic replacement algorithm called VBQ(VDI-Based Queues) to manage the SSD-cache based on the hotness of VDIs(Virtual Disk Image). The hotter virtual disks deserve more allocated space in the SSD-cache. We also propose a novel replacement policy to manage SSD-caches by dynamically manipulating the upper bounds and lower bounds. To provide services for thousands of clients, there are thousands of virtual disks. The policy adjusts the available storage space of every queue dynamically based on the real-time access number of virtual disks. To reduce the impact of destaging on overall performance, the dirty data in the cache will be written back to data nodes during idle time. At the same time, we take advantage of the redundant storage in the data nodes as logging areas to retain the original data reliability of the system.

Our contributions of this paper are:

- We designed a multi-tiered distributed storage system called CircularCache, which supports hot data distribution across multiple cache nodes. The CircularCache uses consistent hashing to distribute hot data across multiple cache nodes.
- For cache management, we exploit a replacement policy called VDI-Based Queues(VBQ). To retain high performance of SSD-caches, a novel replacement policy is designed by dynamically manipulating the upper-bound and lower-bound of queues.
- The CircularCache is implemented based on Sheepdog [25]. Experimental results show that our CircularCache offers a performance improvement by up to 270% under various workloads compared with the original distributed storage system without SSD-cache.

The outline of this paper is organized as follows. Section 2 presents related works. Section 3 and Section 4 show the design and implementation of CircularCache. Section 5 conducts experiments. Section 6 concludes this paper.

II. Related Work

A. Replacement policy

Recency-based replacement policies are developed based on LRU, which is one of widely used replacement policies. LRU utilizes temporary locality(recency) to decide which cold data to be evicted. This policy has low overhead, but it may encounter the cache pollution problem(for example, accessing a large stream will evict large amount of hot data from cache). To remedy this problem, several improved replacement policies take the access frequency into account, such as LRU-k [1], 2Q [2], Multi-queues [3]. LRU-k only holds the data items whose recently access number are more than k. 2Q uses a FIFO queue and an LRU queue to filter the evicted data. For Multi-queue, there are many LRU queues which are in the order of access recency and the data item which is least recently accessed will be evicted at the highest priority.

Frequency-based replacement schemes are based on LFU, within which least frequent accessed data are at the highest priority to be evicted. The data is ordered based on the number of accesses. While LFU only concerns access frequency, there are two shortages of LFU: maintaining large amounts of accessing information and slowly reacting to frequency changes. In order to reduce the overhead, LFU* is proposed to only record the data items those are accessed only once. In-memory LFU [5] only record the access number of data are already in cache. In order to exploit decay mechanism, LFU-Aging [6] introduces the recency and Window-LFU [4] only records the last W data items.

FIFO-based replacement schemes are FIFO, within which the data which firstly enter will be evicted firstly. The limitation of FIFO is that it does not considered frequency. FIFO-based replacement policies include Clock [7] and Second Chance. Clock considers the temporal locality and spatial locality. The data is ordered in LBA and a flag is used to record recency. Second Chance gives some data second chance for reuse.

B. SSD-based cache management

There are two kinds of hierarchies to combine SSDs and HDDs: the vertical hierarchy and the horizontal hierarchy. For vertical hierarchies, SSDs and HDDs are organized as a tiered storage. Most of these schemes use SSDs to store hot data and HDDs to store cold data. I-CASH [11] uses SSDs to hold reference blocks that update infrequently. For horizontal hierarchies, due to high-performance of SSDs and high-capacity of HDDs, it is popular to deploy SSDs as a cache above HDDs. Sieve-store [8] uses global shared SSDs to enhance performance. In order to reduce the overhead of maintaining data accessing history, Hystor [9] exploits a scheme called block table [10], which is a three-level tree structure, to achieve it at low overhead.

Because one shortage of SSDs is the lifetime, many algorithms [14] [15] [16] [17] [18] [19] are designed for SSD cache to reduce the negative impacts of write amplification. CFLRU [12] splits the cache into two areas: the working-region and the clean-region. The clean data in the clean-region are evicted at the highest priority. LRU-WSR [13](Write Sequence Reordering) reduces the write by reordering write requests.

There are some algorithms utilizing the NVM inside SSDs to decrease write amplification. Flash Aware Buffer(FAB) [19] manages the collected pages within the block. However this algorithm does not consider the recency. Coldest and Largest Cluster(CLC) [20] can remedy this problem. In addition, Block-Page Adaptive Cache (BPAC) [21] has been proposed to improve the CLC algorithm.

III. Design

A. Motivations of CircularCache

SSDs have attracted a large amount of attentions because of its many advantages, such as low-energy, low-latency, high-throughput, and high durability. SSDs are too expensive so far compared to HDDs, so it can not replace HDDs in a few years. SSD-based arrays have been applied to architect high-performance storage systems, but the high-price and low-capacity have limited its usage. In addition, the cold data account for a large fraction of total data, which will waste the capacity of SSDs. Using SSDs to host cold data will reduce cost-performance. SSD-based caches are usually deployed as a cache to enhance performance, where SSDs hold hot data and HDDs hold cold data.

For SSD-based caches, the manners to employ SSDs fall into two categories: ensemble-level cache schemes and per-server cache schemes. For per-server cache schemes, the hotness of servers are imbalance. Some cold servers only have small amount of hot data, so the cache inside these servers will not make sense. As hot nodes deserve more cache space, the ensemble-level cache can be flexibly shared by servers.

With the sharply increasing amount of data, the cache size requirements are increasing. The critical issue is to construct a scalable cache. When the global cache is organized by several cache nodes, it is challenging to distribute data evenly and to balance workload. In addition, the associated replacement policy should consider not only the recency and frequency but also the characteristics of access patterns in virtual environment.

To construct a scalable cache, we use consistent hashing table to distribute data. For shared storages, the global cache is organized by several SSD cache nodes. We refer to this tiered distributed storage system as CircularCache. As shown in Figure 1, the SSDs and HDDs are organized in horizontal hierarchy. The upper layer is the SSD-cache organized by multiple SSD nodes. The lower layer is a block-level storage system. Unlike the file-level systems [27] [26], the block-level storage systems, such as Dynamo [23] Sheepdog [25] and Cassandra [24], can support heterogeneous client applications.

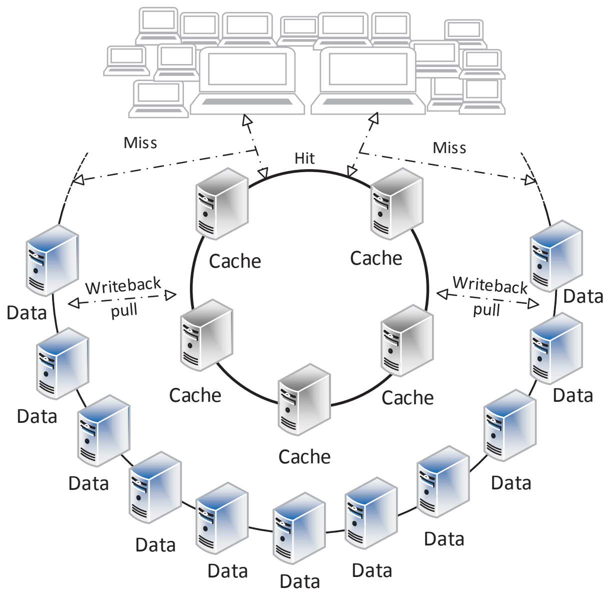


Fig. 1. The Architecture of CircularCache

B. Cached data layout

To balance the workloads among cache nodes, we utilize the consistent hashing table to distribute the data. The advantage of consistent hashing table is that when a new node is added or an old node is removed, the amount of migrated data is very small. All nodes are organized on a ring where every node manages a contiguous range of keys. For every data item, it is assigned a key. The key will be hashed by dedicated hashing functions. If the replication factor is N, the data will be held in the first N nodes which are encountered on the ring.

For each physical node, it is in charge of one contiguous range on the ring. With virtual nodes, the partition policy is changed from one range per node scheme to many sub-ranges per nodes scheme. Virtual nodes, which is used to offer better load balance guarantee, is introduced in Dynamo [23] and Cassandra [24]. The virtual nodes can make the system rebuild failed nodes faster than one-range per node scheme. In addition, virtual nodes can make system allocate the proportional storage space based on the capacity of heterogeneous servers. In other words, the server with larger capacity will be in charge of larger range on the ring.

Figure 2 presents data layout of CircularCache and processes for adding/removing cache nodes. Every physical node is mapped to several virtual nodes. When the keys of data $x1$, $y2$, $z2$, and $w1$ are between $(A1, B1]$, these data items are sent to the first N virtual nodes encountered that belongs to different physical nodes. Figure 2 also shows physical node B is removed and physical node D is added. When physical node B is unavailable, keys which originally live in virtual node $B1$ are adjusted to the next virtual node $C1$. Thereby, data items $x1$, $y2$, $z2$, and $w1$ are migrated to physical node C. When the physical node D is added into this ring, the range of keys belong to $A3$ is partitioned into two subranges, one of which belongs to virtual node $D1$. The data items $y1$, $w2$ are migrated to physical node D.

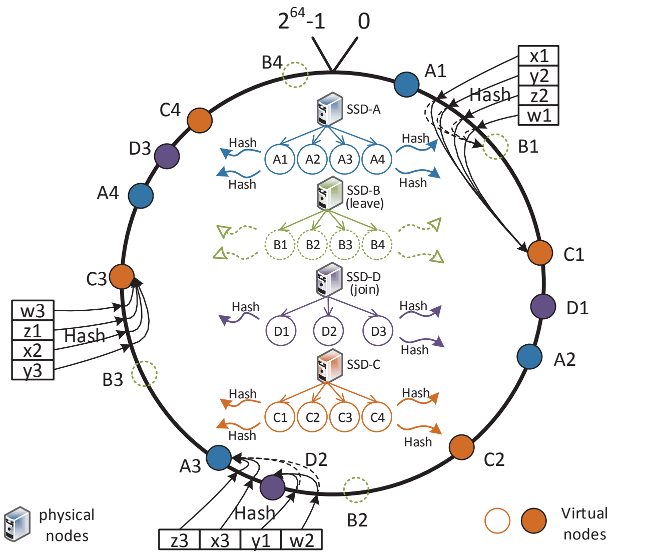


Fig. 2. Data Layout of CircularCache and processes for removal of node B and addition of node D

C. VDI-Based Queues

IaaS tends to use block-level interfaces to adapt to heterogeneous clients. Unlike file-level systems, block-level distributed storage systems have difficulty in utilizing sematic analysis. However, for virtual environment, there are several attributes: VDIs, clients, and directories. Our VBQ(VDI-Based Queues) algorithm considers not only the access interval but also access numbers. The hotter VDIs deserve to be allocated more free space. In other words, the data items of shortest queue are at the highest priority to be evicted.

Using virtual disks, the physical storage resources can be shared flexibly. To provide services for thousands of clients, there are thousands of virtual disk. The cached data items are stored in several queues based on virtual disks. The hot data of the same virtual disk are managed by a queue. The algorithm adjusts the available storage of every queue dynamically based on the access number of every virtual disk. In order to obtain better performance, an upper bound and a lower bound are set for every queue, and all bounds are dynamically adjusted based on access frequency of VDIs.

As shown in Figure 3, the VBQ policy plays an important role in the SSD-cache. The requests are distributed based on consistent hashing table. And the data path of requested is shown as follows. For write requests, because we employ write-back policy for cache, the data will be written into SSDs no matter the write request is hit or not, and N-1 replicas will be sent to corresponding logging areas. For read hits in SSDs, the read request is sent to the corresponding queue. For read miss in SSDs, the read request will be sent to one of associated data nodes randomly.

The maximum length of queue is decided by the total access frequency of the corresponding VDI. As the total access frequency is dynamically varied, the maximum length is dynamically variable. As shown in Figure 3, we assume that there are n VDIs, respectively named VDI_1, VDI_2, ..., and VDI_n. For associated VDIs, the access frequency of each VDI is A_1, A_2, A_3, ..., A_n. These parameters meet the constraint as follows.

$$A_1 \geq A_2 \geq A_3 \geq ... \geq A_n \tag{1}$$

The total access frequency of all VDIs is $count$ and the total size of the SSD-cache is S, so the maximum length of $VDI_m(0 < m < n+1)$ is calculated as the following equation.

$$L_m = A_m * S/count \tag{2}$$

where A_i is referred to as the total access frequency of $VDI_i(0 < i < n+1)$.

Considering the shortage of SSDs in terms of lifetime, we should reduce the number of write operations. Our VBQ achieves this goal by reducing the amount of cold data written into SSD. Thereby, we use a sieve-LRU queue to filter the cold data, the length of this queue is fixed and the sieve-LRU only store the metadata of requested data. The filtering condition of entrance into VBQ is that the access number of data items must exceed a predefined threshold. For hits in sieve-LRU, if they meet the condition of entrance into VBQ, the requested data is written into SSD-cache. Otherwise, the request will be sent to corresponding data nodes. For misses in sieve-LRU, the access metadata will be written into sieve-LRU. These access metadata are maintained in the sieve-LRU.

D. Replacement policy

As we use a hash-based cache, the critical issues include: when to start eviction, which to evicted and how many to be evicted. We use two variables(the upper-bound and the lower-bound) to decide which queue to start eviction and when to stop eviction. We use upper bounds to trigger to start eviction and use lower bounds to stop eviction. When the used capacity of the SSD-cache has exceeded a predefined global threshold, the cache begins to reclaim automatically. When the available space have exceed a predefined global threshold, the reclaim processing is stopped.

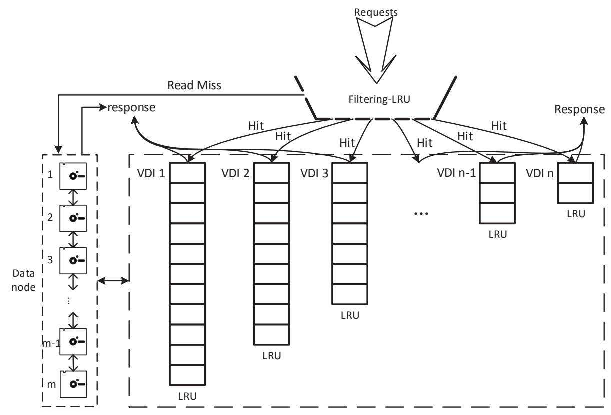


Fig. 3. VDI-based Queues

Traditional High-Low Water Mark(HLWM) [29] algorithm uses the high-water mark to decide when to start reclaim and uses the low-water mark to manipulate the amount of data to evict. However, the thresholds of HLWM algorithm are fixed, which will not adapt to the various workloads. Adaptive HLWM [30] can dynamically manipulate the thresholds. However, our VBQ is designed based on multiple queues. For every queue, there are an upper bound and a lower bound to decide when to start eviction and stop eviction.

The range of upper-bounds is $[U_{min}, U_{max}]$, and the range of lower-bounds is $[D_{min}, D_{max}]$. These parameters should meet the constraint as follows.

$$1 > U_{max} \geq U_{min} \geq D_{max} \geq D_{min} > 0 \tag{3}$$

While the U_i represents the upper-bound of i_{th} queue

$$U_i = \begin{cases} U_{min} + \frac{A_1 - A_i}{A_1 - A_n} * (U_{max} - U_{min}) & \text{if } A_1 \neq A_n \\ U_{max} or U_{min} & \text{if } A_1 = A_n \end{cases} \tag{4}$$

While the D_i is referred to as the low-bound of i_{th} queue.

$$D_i = \begin{cases} D_{min} + \frac{A_1 - A_i}{A_1 - A_n} * (D_{max} - D_{min}) & \text{if } A_1 \neq A_n \\ D_{max} or D_{min} & \text{if } A_1 = A_n \end{cases} \tag{5}$$

The principles of our reclaim policy are shown as follows.

(1) For upper-bounds, the colder queues have higher upper-bounds and the hotter queues have lower upper-bounds. Firstly, this can prevent short queues to be evicted frequently, thus it can promote fairness for all queues. Secondly, the longer queues tend to hold more hot data, the free spaces of long queues are easy to deplete. Thereby, the policy that makes hotter queues have lower upper-bound can alleviate the traffic caused by intensive eviction operations.

(2) For lower-bounds, the colder queues have lower low-bound and the hotter queue has higher low-bound. In this way, the hot data have more opportunities to stay in SSD-cache.

(3) The amount of reclaimed data in colder queues is greater than in hot queues. In other words, the upper bounds and lower bounds meet the condition of

$$Un - Dn \geq ... \geq U3 - D3 \geq U2 - D2 \geq U1 - D1. \quad (6)$$

In this manner, the hotter VDIs can be allocated more space in SSD-cache. Furthermore, this can promote fairness by narrowing the gap between short queues and long queues.

One critical issue is to control the beginning and stoping of reclamation. We use a global upper-threshold ξ and a lower-threshold $\alpha(\alpha < \xi)$ to manage the reclamation. When the used capacity of SSDs exceeds the global threshold ξ, the start condition of reclamation is triggered. When the used capacity of SSDs is under global threshold $\alpha(\alpha < \xi)$, eviction is stopped. While the variable ξ represents the global upper threshold, every queue has its local upper threshold U_i. If all queues have not exceeded the upper-bound of eviction, the used capacity will not exceed the global upper threshold ξ. Thus, the ξ and U_i should meet the constraints as follows.

$$U_1 * L_1 + U_2 * L_2 + U_3 + L_3 + ... + U_n * L_n \leq \xi * S \quad (7)$$

where L_i represents the maximum length of i_{th} queue and S is the total size of SSD-cache. After simplification, the equation is shown as follows.

$$U_{min} + \frac{U_{max} - U_{min}}{A_1 - A_n} * (A_1 - \frac{\sum_{i=1}^{n} A_i^2}{\sum_{j=1}^{n} A_j}) \leq \xi \quad (8)$$

where A_i defines the access number of i_th queue. If A_i is equal to A_n, the equation is satisfied.

For our VDI-base queues, another critical issue is the eviction order. As shown in Figure 4, we present an example of eviction order. Assuming that there are 5 VDIs, the access number of VDI_1 is the largest and the access number of VDI_5 is the least. $D_1 - D_5$ and $U_1 - U_5$ both meet the rules as shown in the previous equations. At first, the data in queues of VDI_5 is evicted first, then data in queue of VDI_4 and VDI_2. Because the amount of data in VDI_3 has not exceeded the upper-threshold U_3, this queue is not required to be evicted at this time.

E. Destage

Destage schedulers are used to flush the dirty data from SSD-caches to HDDs. The critical issues of destaging also include: when to start destaging, how much to be destaged, and which dirty data to be destaged. In order to reduce negative impacts of destaging operations on the overall performance, the main idea of our destage algorithm is to process destaging operations during idle time.

The destaging opportunities are determined by the time-varying I/O workloads. When the workload is light, the start condition of destaging process is triggered. When the workload becomes heavy, the destaging process is terminated. Thus, it can reduce the negative impacts of destage on overall performance.

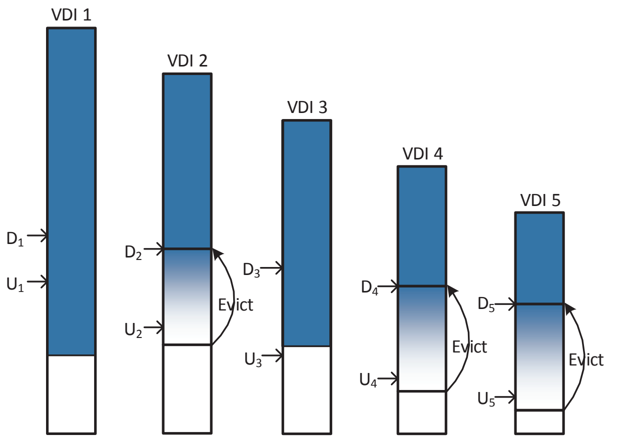


Fig. 4. Eviction of cold data

The destage order is very important. Because the destaging processes are the preparation processes for data eviction. Thereby, for our VDI-based queues, the queue which is at highest priority to evict will be destaged at first.

F. Logging management

For dirty data in SSD-caches, when the SSD nodes failed, this may result in data loss. We have to make replication for dirty data to retain reliability.

There are two alternative logging management schemes: using extra dedicated nodes as logging area and using the free space in data nodes. In order to utilize the high performance of SSD-caches and reduce financial costs, we use the free space of data nodes as logging areas. For N-way replicated distributed storage system, the SSD nodes only store one copy, the remaining N-1 copies are stored in data nodes. Every node has some redundant space, so we can utilize these free space to organize the logging area.

For write requests in SSD-caches, the traditional scheme is to distribute the data in the first N nodes encountered in the ring. However, within our CircularCache the logged data will not stored in the original nodes. In Figure 5 for example, for 3-n way replication, if the copies of data items are distributed in $D1$, $D2$, and $D3$. To retain high reliability, the logging data of this data items will not be held in $D1$, $D2$, and $D3$. As shown in Figure 5, when the requested data is hit in cache node $S2$, one copy will be written into $S2$ and two other copies will be written into data nodes $D4$ and $D5$.

Two cases will trigger the reclamation of logged data. First, when the destage of SSD nodes is triggered, dirty data will be written into corresponding data nodes and be deleted from SSD-caches. Second, when the amount of data in logging area exceeds a predefined threshold, the system begins to reclaim the dirty data.

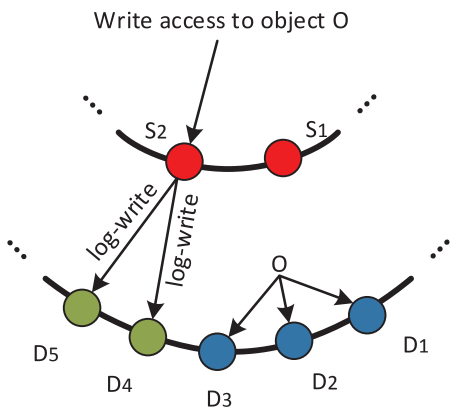


Fig. 5. Logging management

G. Fault tolerance and recovery

For storage systems, there are two alternative manners to retain high reliability. First, erasure-coding schemes can save space but they introduce extra overhead. Second, replication schemes require large amount of storage space. As the erasure-coding scheme requires extra time to encode and decode for data, which will introduce negative impacts on performance. Within CircularCache, there are two kinds of nodes: SSD-nodes and data nodes.

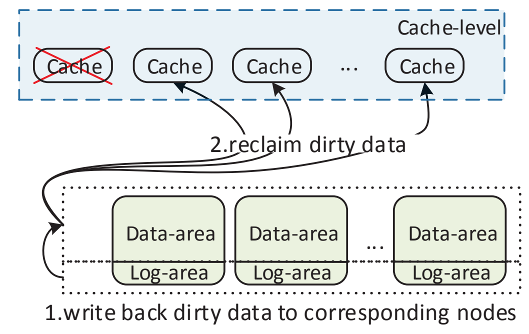
For SSD node failures, there are two logging data to backup the dirty data. As the level of fault-tolerance for dirty data is the same to the normal data. There are N replicas for clean data, thus the clean data do not need recovery. The storage system can be recovered the lost data from logging nodes. However, the dirty data will be written back to data nodes. In order to reduce the data migration, the dirty data items logged in data nodes are claimed back to data nodes. The recovery process for SSD node failures is shown in Figure 6(a).

For data node failures, the redundant data can offer data reliability guarantee. For N-replication storage systems, it can tolerance at most $N - 1$ nodes failure. For logging data, the logged data are stored in data nodes. Unlike the normal data, the logging data will be written back into corresponding data nodes. Because the dirty data will be eventually written back to corresponding data nodes, they do not require to be recovered. The recovery process for data node failure is shown in Figure 6(b).

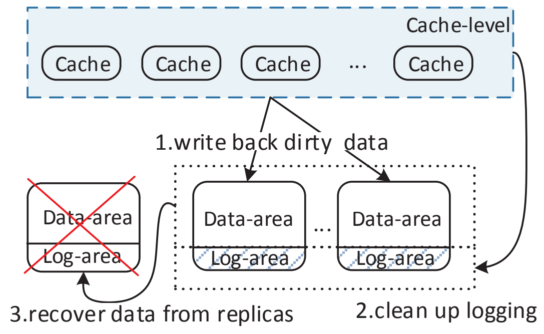
IV. Implementation

We implement CircularCache based on a block-level distributed storage system called Sheepdog, which is designed for volume management and container service. The advantages of Sheepdog include easy use, high scalability, and simple coding. Sheepdog works with QEMU [28], emulating virtual devices, as well as corosync, maintaining membership of nodes.

As shown in Figure 7, CircularCache relies on three critical modules, including Requests Distribution, Cache Manager, and Logging Manager. Requests Distribution, modified from



(a) SSD Node Failure



(b) Data Node Failure

Fig. 6. Reliability guarantee

Sheepdogs original module - Gateway, controlling the data path of requests. In other words, this module decides which node to service the requests based on consistent hashing table. Cache Manager maintains VBQ(VDI-Based Queue) by analyzing access records of objects to evict cold data and destage dirty data. Sometimes, it gives assistance in recovery. We utilize Logging Manager to retain consistency and reliability by logging writes and information of corresponding cache nodes. When a node, especially a cache node, fails, Logging Manager is in charge of updating data to the latest.

V. Evaluation

A. Configuration

We evaluate CircularCache on a 9-server cluster, where a server is regard as a data node with HDDs or a cache node with SSDs. The parameters of servers are shown in Table I.

TABLE I
The Features of Servers

OS	Fedora 14 (2.6.35.14-106.fc14.x86-64)
CPU	Intel(R) Xeon(R) CPU E5606 @ 2.13GHz * 2
memory	Hyundai 1066 MHz 4GB * 4
network adaptor	Intel Corporation 82574L Gigabit
HDD	ST1000DM003-9YN162
SSD	SanDisk SATA Revision3.0 6Gb/s SSD 64G

In the experiment, we use QEMU to construct three virtual disk images: $Alice1$, $Alice2$ and $Alice3$. As well as, we choose three representative traces, which are collected from

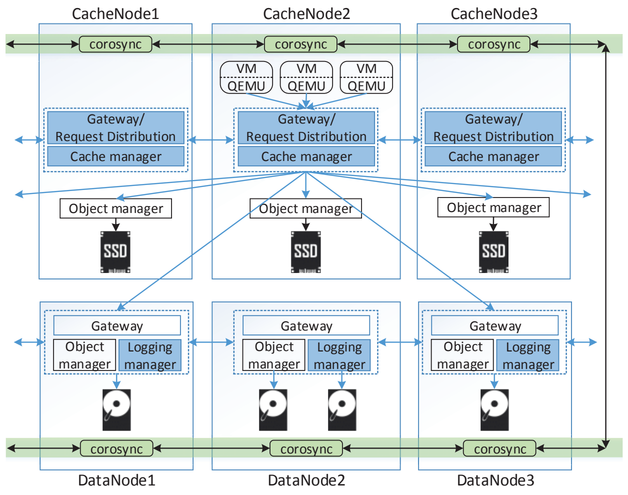


Fig. 7. The Implementation of CircularCache

enterprise servers by Microsoft Research Cambridge [25]. Table II lists the characteristics of traces, such as IOPS and write ratio, and the corresponding VDI of each trace. Means that, using the tool Blktrace, usr trace is replayed on the virtual disk, to which QEMU converts $Alice1$. (At the same time, other traces are replayed on other virtual disks to simulate multi-user scene.) As three traces record week-long real-life workloads, we accelerate the replay of these traces.

TABLE II
THE FEATURES OF TRACES

VDIs	Traces	Write Ritio	IOPS	avg req(KB)
Alice1	usr	59%	83.87	22.66KB
Alice2	rsrch	91%	21.17	8.93KB
Alice3	web	70%	50.32	14.99KB

In our experiment, there are two configurations for comparison.

- Standard Sheepdog: an unmodified Sheepdog without an SSD-cache. This baseline system deploys 6 HDD-based servers as data storage nodes.
- CircularCache: a Sheepdog modified for our proposed scheme. This system deploys an SSD-cache in front of data nodes. We employs 3 SSD-based servers as cache nodes.

B. Performance Impact

Figure 8 illustrates the comparison between performance of two configurations under three different traces. First, under various traces, CircularCache performs better than standard sheepdog. The reason why CircularCache can achieve improvement is that SSDs have higher IOPS and lower latency compared with HDDs. Second, CircularCache under usr trace gets the best improvement(270%), for the least write ratio among all traces. The average response time of standard sheepdog is 450.35ms, while CircularCache's is 121.37ms. The distributed storage system we used is based on N-way replication scheme and strong consistent mechanism. In order to attain strong consistent, writes will be responded when N replicas are written into corresponding nodes. Since N-1 replicas are written to logging-area, the SSD-cache improves performance mainly reflected in reads. Third, we find that CircularCache under $rsrch$ performs better than under usr. The write ratio of $rsrch$ is larger than web, but the workload of web trace is heavier than $rsrch$. Thus, CircularCache gets only 20.67% improvement with web trace.

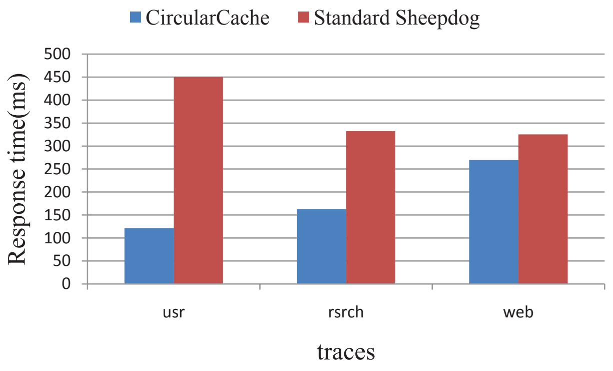
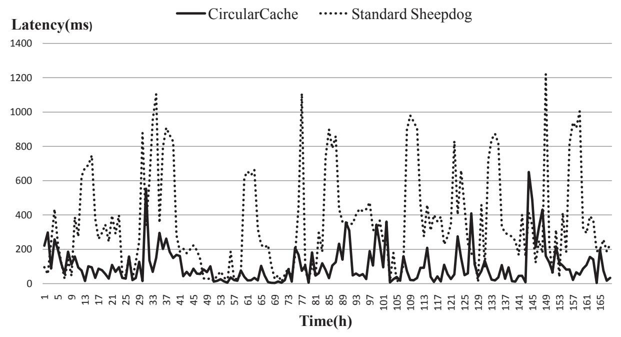


Fig. 8. Total Performance under various traces

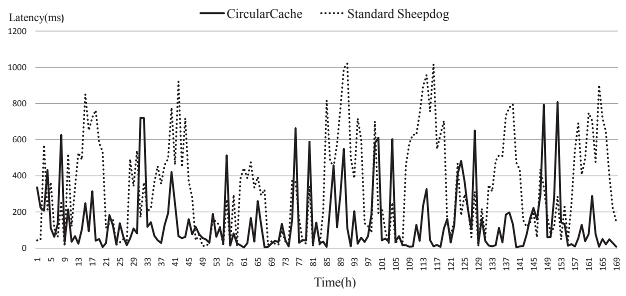
Figure 9(a) shows the response time distribution under usr trace. Throughout 168-hour, the response time of standard sheepdog is basically greater than CircularCache's. Sometimes, such as during [57h, 69h], the difference is more obvious because of high hit ratio in the SSD-cache and high read ratio of workload. Figure 9(b) shows the comparison of the response time between two configurations under web trace. The average response time of CircularCache and standard sheepdog are 269.59ms and 325.32ms respectively. The reason why web trace gets the least improvement is that the write ratio of web is larger than usr and the IOPS of web is larger than $rsrch$. During from 145h to 152h, the CircularCache performs worse than standard sheepdog. This is because large amounts of writes and low hit ratio in SSD-cache can lead to a large amount of reclamation operations, which have a negative impact on overall storage performance.

C. The amount of evicted data and hit ratio

Figure 10(a) presents the amount of evicted data for every VDI-based queues in different cache nodes. First, the total amount of evicted data in every cache node is almost the same, while the difference between every VDI-based queues is obvious. There are two reasons why the amount of evicted data for $Alice1$ with usr trace is the least. One, IOPS of usr is largest than others, which makes the corresponding queue as the last choice when evicting. Second, as shown in Figure 10(b), usr gets the highest hit ratio in the SSD-cache. This means that temporary locality of usr is better than other traces. Second, for VDI $alice2$, the amount of evicted data in SSD-1 is larger than in other SSDs. As cache objects are statically distributed according to consistent hashing, objects in SSD-1 have poor temporary locality. Third, cache objects



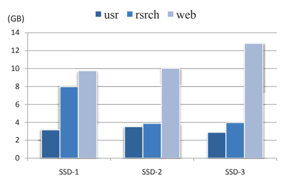
(a) under usr



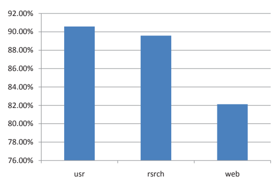
(b) under web

Fig. 9. The average response time under various traces

for $Alice2$ with $rsrch$ can be considered to evict firstly, while its amount of evicted data isn't the largest, which indicates fairness.



(a) The amount of evicted data



(b) Hit ratio

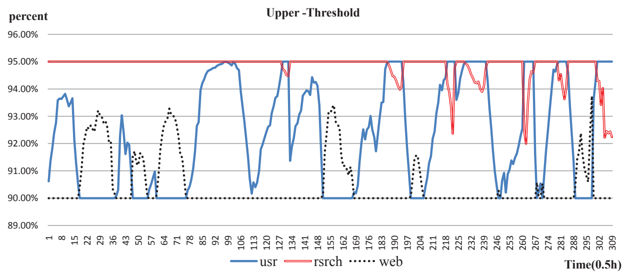
Fig. 10. The amount of evicted data and hit ratio

D. The effect of range

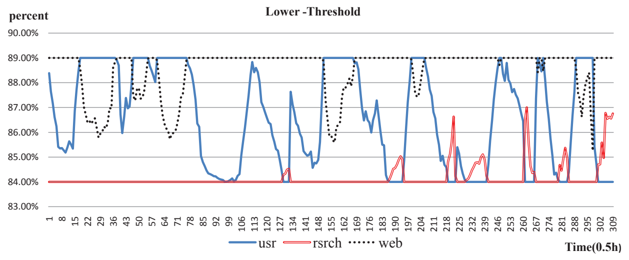
In our evaluation, the range of upper-bounds is set from 90% to 95%. The Figure 11(a) presents the range of upper-bounds under various traces. First, the $rsrch$ workload almost have the highest upper-bound, thereby this means $rsrch$ is the coldest VDI. For upper-bound, the colder VDI has higher upper-bound.this can prevent the short queues to be evicted frequently, thus it can promote fairness of all queues. Secondly, the web trace has the lowest upper-bound, the longer queues tend to host more hot data, the free spaces of long queues are easy to deplete. Thereby, the policy that makes hotter queues have lower upper-bound can alleviate the traffic caused by intensive eviction operations.

The Figure 11(a) presents the range of lower-bounds under various traces. the range of lower-bounds is set between 84%-89%. We can find the $rsrch$ has the lowest. the colder queue has lower low-bound and the hotter queue has higher low-bound. In this way, the hot data have more opportunities to stay in SSD-cache. Secondly, the As the amount of reclaimed data under web trace is largest as shown in previous section.

Through the Figure 11, the amount of reclaimed data in colder queue is greater. In other words, the upper bounds and lower bounds meet the condition of $Un - Dn \geq ... \geq U3 - D3 \geq U2 - D2 \geq U1 - D1$. In this manner, the hotter VDIs can achieve more space in SSD-cache . And this can promote fairness by narrowing the gaping between short queues and long queues.



(a) the fluctuation of upper-bounds under various traces



(b) the fluctuation of lower-bounds under various traces

Fig. 11. The upper-bounds and lower-bounds

As shown in Table III, we use two sets: set1(95-90-89-84) means the range of upper-bounds is from 90% to 95%, the range of lower-bounds is from 84% to 89%. The set2 (95-93-92-90) has the same definition. The ranges of upper-bounds and lower-bounds can affect the amount of evicted data and the resident time in SSD. After we change the range of upper-bounds from 84%-89% to 92%-90%, more hot data will be host in the cache. Thus, the set2 has better performance than set1.

TABLE III
THE IMPACT OF DIFFERENT RANGES

VDI Name	95-90-89-84	95-93-92-90	improvement
Alice1	121.37ms	115.54ms	5%
Alice2	163.09ms	153.39ms	6.3%
Alice3	269.59ms	257.42ms	4.7%

E. The impact of cache size

If cache size of an SSD node is set as 1G, the total cache size account for 18.44%. We changed the size of an SSD node from 1G to 2G, as list in Table IV, the performances under various traces are all improved. With larger cache size, more hot data can be host in SSD-cache and hit ratio will be increased. Thereby, the CircularCache has lower response time than standard sheepdog due to high performance of SSD-cache.

TABLE IV
THE IMPACT OF DIFFERENT CACHE SIZE

VDI Name	SSD-994MB	SSD-1988MB	improvement
Alice1	121.37ms	41.54ms	192%
Alice2	163.09ms	69.82ms	130%
Alice3	269.59ms	115.01ms	134%

VI. CONCLUSION

In this paper, we proposed a scalable and adaptive SSD-based cache called CircularCache, which can support hot data distribution across multiple cache nodes. In order to distribute hot data evenly and balance the workload, this paper utilizes the consistent hashing as the cache partition algorithm. In addition, a dynamic replacement algorithm called VBQ(VDI-Base Queue) has been proposed to manage SSD-cache. The hotter virtual disks deserve more allocated free space in the SSD-cache. The replacement policy manages the SSD-cache by manipulating the upper-bounds and lower-bounds of multiple queues. The policy adjusts the available storage space of every queue dynamically based on the access number of every virtual disk. The experiment results show that our scheme offers a performance improvement within the range of 20%-270% under various workloads compared to the standard Sheepdog without an SSD-cache. The experiment results also demonstrate that our replacement policy works well within the CircularCache.

ACKNOWLEDGMENT

This work is sponsored in part by the National Natural Science Foundation of China under Grant No.61472152, the National Basic Research Program of China (973 Program) under Grant No.2011CB302303, the Fundamental Research Funds for the Central Universities, HUST:2015QN069, and the National Natural Science Foundation of China under Grant No.61432007 and No.61300047.

REFERENCES

[1] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," ACM SIGMOD Record, vol. 22, no. 2, pp. 297-306, 1993.
[2] T. Johnson, and D. Shasha, "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm," in Proc. of VLDB, 1994.
[3] Y. Zhou, J. Philbin, and K. Li, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in Proc. of USENIX, 2001.
[4] G. Karakostas and D. N. Serpanos. "Exploitation of different types of locality for web caches". In Proc. of ISCC, 2002.
[5] Stefan Podlipnig and et al.. "A survey of web cache replacement strategies". ACM Comput. Surv., vol. 35, no. 4, pp.374-98, Dec. 2003.
[6] M. Arlitt, R. Friedrich, and T. Jin, "Performance evaluation of web proxy cache replacement policies," Perform. Eval., vol.39, no.1-4, pp. 149-64, 2000.
[7] A. J. Smith. Sequentiality and Prefetching in Database Systems. ACM Transactions on Database Systems, 3(3):223C247, September 1978.
[8] T. Pritchett, and M. Thottethodi, "SieveStore: a highly-selective, ensemble-level disk cache for cost-performance," ACM SIGARCH Computer Architecture News, vol. 38, no. 3, pp. 163-174, 2010.
[9] F. Chen, D. A. Koufaty, and X. Zhang, "Hystor: making the best use of solid state drives in high performance storage systems," in Proc. of ICS, 2011.
[10] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang, "DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities," in Proc. of FAST, 2005.
[11] J. Ren and Q. Yang, "I-CASH: Intelligently coupled array of ssd and hdd," in Proc. of HPCA, 2011.
[12] S. Park, D. Jung, J. Kang, J. Kim, and J. Lee, "CFLRU: a replacement algorithm for flash memory," in Proc. of CASES, 2006.
[13] H. Jung, H. Shim, S. Park, S. Kang, and J. Cha, "LRU-WSR: integration of LRU and writes sequence reordering for flash memory," IEEE Trans. Consumer Electronics, vol. 54, no. 3, pp. 1215-1223, 2008.
[14] J. Hu, H. Jiang, L. Tian, and L. Xu, "PUD-LRU: An erase-efficient write buffer management algorithm for flash memory SSD," in Proc. of MASCOTS, 2010.
[15] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: a flash translation layer employing demand-based selective caching of page-level address mappings," in Proc. of ASPLOS, 2009.
[16] Z. Fan, D. H. Du, and D. Voigt, "H-ARC: A non-volatile memory based cache policy for solid state drives," in Proc. of MSS, 2014.
[17] C. Wang, et al, "NVMalloc: Exposing an aggregate SSD store as a memory partition in extreme-scale machines," in Proc.of IPDPS, 2012.
[18] T. Kgil, and T. Mudge, "FlashCache: a NAND flash memory file cache for low power web servers," in Proc. of CASES, 2006.
[19] H. Jo, J. Kang, S. Park, J. Kim, and J. Lee, "FAB: Flash-aware Buffer Management Policy for Portable Media Players," IEEE Trans. Consumer Electronics, vol. 52, no. 2, pp. 485-493, 2006.
[20] S. Kang, S. Park, H. Jung, H. Shim, and J. Cha, "Performance Trade-offs in Using NVRAM Write Buffer for Flash Memory-based Storage Devices," IEEE Trans. Computers, vol. 58, no. 6, pp. 744-758, 2009.
[21] G. Wu, X. He, and B. Eckart, "An Adaptive Write Buffer Management Scheme for Flash-based SSDs," ACM Transactions on Storage (TOS), vol. 8, no. 1, pp. 1:1-1:24, Feb. 2012.
[22] E. K. Lee, and C. A. Thekkath, "Petal: Distributed Virtual Disks," in Proc. of ASPLOS, 1996.
[23] G. DeCandia, D. Hastorun, M. Jampani, and et al.. Dynamo: Amazons highly available key-value store. In Proc. of SOSP, 2007.
[24] A. Lakshman and P. Malik, "Cassandra - A Decentralized Structured S-torage System," In Proc. of the 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, 2009.
[25] P. Maciel, et al, "Performance evaluation of sheepdog distributed storage system," in Proc. of SMC, 2014.
[26] P. J. Braam, and R. Zahir, "Lustre technical project summary," Cluster File Systems, Inc., Mountain View, CA, Technical Report, 2001.
[27] S. A. Weil, S. A. Brandt, E. L. Miller, D. E. L. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in Proc. of OSDI, 2006.
[28] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in Proc. of USENIX, 2005.
[29] A. Varma, and Q. Jacobson, "Destage Algorithms for Disk Arrays with Nonvolatile cache," in Proc. of ISCA, 1995.
[30] Y. J. Nam and C. Park, "adaptive high low water mark destage algorithm for cached raid5," in Proc. of PRDC, 2002.