SSD as a Cloud Cache? Carefully Design about It

Yi Liu[1,3] Xiongzi Ge[2] David H.C. Du[2] Xiaoxia Huang[1,3]

[1] Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences
Shenzhen 518055, P.R. China
{liuyi,xx.huang}@siat.ac.cn

[2] Department of Computer Science, University of Minnesota
Twin Cities, USA
{xiongzi,du}@cs.umn.edu

[3] Shenzhen College of Advanced Technology, University of Chinese Academy of Sciences
Shenzhen 518055, P.R. China

Received 9 December 2014; Revised 4 January 2015; Accepted 18 May 2015

Abstract. Virtualization in clouds is promoting the current trend of sharing the storage with multiple tenants. This brings us two fundamental design issues when considering SSDs as a shared storage cache. (i) How can we choose the hierarchy cache model to reduce I/O latency? (ii) How can we design the dynamic cache space allocation strategy to maximize utilization of SSDs space? This paper mainly proposes the corresponding solutions to address the above two issues. (i) We design a cache-awareness model to avoid the useless network latency for querying. (ii) With using the weighted max-min fair share algorithm, measuring the weighted value of each tenant through recording the states of four multi-dimensional factors, can be beneficial to make wise decisions on SSD space allocation. Our experimental results validate that the cache-awareness model outperforms the other two models (cache-unawareness model and without cache) by 1× to 4× in latency. Meanwhile, compared to the static weighted value initializations of the max-min algorithm, our method with dynamically measuring weight value on a tenant basis can achieve much better space utilization.

Keywords: SSDs, cache model, weighted max-min fair share

1 Introduction

It has been predicted that the amount of digital data, which is either stored or processed in cloud-based data centers, is expected to double annually in 2020 [3]. High-performance I/O intensive data processing is in an irreversible high demand trend. Although I/O parallelism technologies, like typical RAID, have been widely used, the I/O speed of Hard Disk Drives (HDDs), the preferred permanent back-up storage media, is still restricted by the mechanical movement for track seeking operation. Specially, data transfer started from HDDs is considered as one of the unpredictable performance bottleneck in data centers [4]. NAND Flash-based Solid State Drives (SSDs) are becoming a nature choice to speedup disk I/O throughput [5]. Rather than HDDs completely displaced by SSDs, placing SSDs as a tier-2 read cache for HDDs (the tier-1 cache is DRAM) is a common solution to balance the cost-effective [6-9]. In a nutshell, the performance per cost of SSDs is in the middle of RAM and HDDs.

Dm-cache mechanism [22] and its derived client-side SSD caching techniques [7, 9] have been well studied. Generally speaking, SSD is directly deployed as a tier-2 cache on the client side for storing the most-likely-to-be-reused data in order to avoid network RTT delay of data transmission as possible. As part of the Linux kernel, dm-cache is a useful tool which allows fast one or more SSDs to speed up I/O performance. However, such dm-cache method is not a cloud-based solution in which each client is assigned a predefined fixed-size cache. This method can be regarded as a decentralized resource management without elastic running adjustment for QoS providing, live VM migration, etc. Meanwhile, vCacheShare [23] complexly adopts a cache utility model for dynamic resource allocation without using the widely used and simple max-min fairness resource allocation strategy [32]. Moreover, vCacheShare is lack of the further discussion in performance effects of choosing different caching models.

This paper focuses on effectively using SSD as the tier-2 cache in cloud environments to strengthen the past client-side SSD caching techniques. To get an efficient design, we face the following two critical challenges:

i. The cache model must be carefully redesigned to accommodate with both the SAN environment and multi-level cache hierarchy. Traditionally, the cache-unawareness model has been widely used in multi-level cache hierarchy [13, 14] especially in the top-to-down L1-L3 CPU cache [15]. The “unawareness”

indicates that the client does not know whether the data request can be satisfied from the tier-2 cache or not. We found the cache-unawareness model may become potential bottleneck for improving I/O throughput under two facts: (1) Access patterns in the tier-2 cache are different from the tier-1 cache, because the tier-2 cache only receives those requests which have been missed in the tier-1 cache. (2) The extra transmission I/O latency of SAN is unavoidable under the condition that the hit ratio in the tier-2 cache is low.

ii. The dynamic cache space allocation policy for SSD pooling are influenced by multiple dimensions including the configured tenant priority, I/O access characteristics, time-varying I/O block request speed, hit ratio, and so on. D. Shue et.al. [16] have pointed out guaranteeing the isolation and the fairness for multi-tenant cloud storage are robustness to skew and shift in tenant demand. However, the dynamic resource allocation makes the cache size of each- either tenant or VM node unpredictable.

These challenges motivate us to rethink the choices of: (1) the SSD-based tier-2 cache model in clouds. (2) the strategy of dynamic cache space allocation. Our main improvements of this paper are listed as follows:

i. A content-awareness SSD cache is built based on the local VM node cache index. Compared with the cache-unawareness model, the key difference is that the index can identify whether a block reading request can be satisfied from its SSD cache region. If not, the block is directly read from its *VMDK* of the HDD pool bypassing the SSD cache. Thus, the extra useless and expensive I/O path through SAN in querying SSD cache can be saved. In order to saving memory overhead of building index, we propose an improved Bloom Filter (BF) [17] as the space-efficient data structure.

ii. The dynamic cache space allocation policy is originated from the well-known weighted max-min fairness share policy on the per-tenant basis. Our improvements include: (1) how to measure the weight value of each tenant through recording states of the multiple dimensions to make better use of SSD space, (2) dividing the shared SSD cache pool into a group of fine-grained quota size for simplifying SSD space management. The quota size is the fundamental unit to be allocated or be reallocated on demand in a either tenant or VM basis.

We implemented our prototype through a trace-driven simulator and performed extensive evaluation using a well-known data center traces to validate that: (1) the cache-awareness model for SSD cache performs better than the other two (cache-unawareness model and without cache) by about 1× to 4× in latency. (2) Through our dynamic SSD cache allocation policy, the overall hit ratio is improved by about 10% to 20%, compared with that of the other three static allocations, respectively.

The rest of this paper is organized as follows. Section 2 summarizes the related work. Section 3 overviews the design issues of SSD cache in clouds. Section 4 gives our SSD cache model and designs. In Section 5, we present the performance evaluation methods and discuss the experimental results. Finally, we conclude this paper in Section 6.

2 Related Work

We categorize and compare previous great related work with our solution as following two aspects.

1. SSD as a Cache. Guided by the performance of Flash-based SSDs which is between DRAM and HDDs, cache management on SSDs has been proposed for speeding up the accesses to disk storage in clouds. Sun's ZFS [20] can treat SSDs as the tier-2 cache under the memory in distributed storage systems. Dm-cache [22] is a generic block-level disk caching utility for distributed storage systems. Flashcache [21], proposed by Facebook, is deployed in many productive cloud systems. vCacheShare [23] is a Server Flash Cache (SFC) in a virtualization environment to solve the real-time resource management problems, traditionally addressed by using heuristics. Mercury [7], a persistent, write-through host-side cache for Flash memory, is designed to accommodate with the trend that data center architects prefer to deploy shared storage over shared direct-attached storage (DAS). Appuswamy et al. [26] proposed to keep exclusive in multi-level caches with SSDs as the second level not only improve performance but also prolong SSD lifetime by reducing the write number of allocations. MOLAR [6] tracks the block demoted counts to decide which blocks are qualified to cache in SSD to reduce useless allocation-writes, meanwhile, grouping the cached blocks to make full use of the advantage of SSDs as possible. Janus [24] is a smart system that makes use of long-term workload access behavior to determine the partition allocations between each workload in a cloud-scale distributed file system. Narayanan et al. [25] analyzed several typical workload traces of data-center servers to evaluate the cost effectiveness of replacing disk based storage by SSDs. They suggested that using SSDs in various I/O applications should be carefully evaluated due to the low capacity per dollar and asymmetric read/write performance.

2. Resource Allocation in Clouds. Cloud platforms (e.g. Openstack [30]) typically allow multiple users, or tenants, to share hardware resources through virtualization techniques (e.g. KVM). Many solutions focus on how to dynamically and fairly allocate resource on a tenant basis to guarantee *SLA*s and achieve high utilization. The max-min fairness [32], which maximizes the minimum allocation received by a tenant, is widely used in strategies of the shared resource allocation. For instances, the Weighted Fair Queuing (WFQ) algorithm [33] is ap-

plied for bandwidth competition in multi-flows and *Dominant Resource Fairness* (DRF) [31], a fair sharing model ensures that generalizes max-min fairness to multiple resources in each tenant. D. Shue [16] seeks to provide system-wide per-tenant weighted fair sharing and performance isolation in multitenant, key-value cloud storage services. vCacheShare [23] is a dynamic, self-adaptive framework for automated server flash cache space allocation in virtualization environments.

Leveraging concepts of the previous related literatures, our improvements include: (1) The previous work is insufficient to further talk about how the interconnection latency in SAN affects the choice of caching model. (2) Although there are many good literatures on exploring the shared storage allocation in clouds (e.g. [24]), multiple dimensional factors should be taken into account to reasonably partition the shared SSD cache for achieving high space utilization.

3 Design Issues

3.1 SSD as a Cloud Cache

NAND Flash memory based SSDs have been revolutionizing from high-end laptops to enterprise data storage systems. Unlike traditional HDDs, a SSD is purely composed of a group of semiconductor chips rather than mechanical moving parts, providing such as low I/O latency, low power efficiency, shock resistance and especially extraordinary speed of random read accesses. Moreover, nearly all SSDs on the sale market support universal host interfaces, such as Serial Advanced Technology Attachment (SATA), following those common interface guarantees transparency between SSDs and HDDs. These positive characteristics have made SSDs as an optimal and potential technology for replacing HDDs. However, from Table 1, we see that the average I/O performance of SSDs is better than that of HDDs but still worse than that of DRAM. Meanwhile, the average cost per GB of SSDs is also in the middle, which is much cheaper than DRAM but more expensive than HDDs. Even considering the price-drop trend, the average cost per GB of SSDs is still much more expensive than HDDs. Moreover, SSDs are a kind of a green-energy device whose power consumption is the lowest. Thus, we believe rather than completely replacing the existing HDDs-based storage systems, integrating SSDs as a tier-2 cloud cache under DRAM-based cache (the tier-1), is cost-effective.

Table 1 Sample characteristics of three typical storage media

Drive	Latency			Power Consumption	Cost ($/GB)
	Read	Write	Erase		
DRAM	35-75 ns	35-75 ns	N/A	2-10 W	6
SSD	60 µs	120 µs	2 ms	0.06- 2.6 W	1.8
15K RPM / HDD	1-5 ms	1-5 ms	N/A	5-18 W	0.42

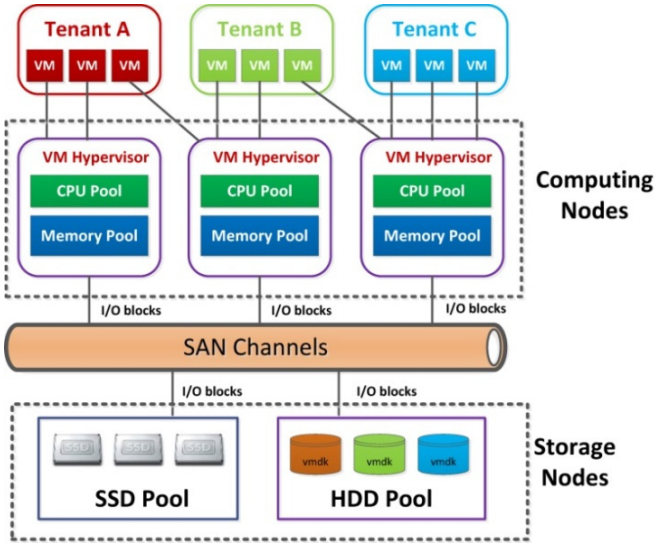

Fig. 1. A simplified architecture for resource virtualization in a cloud-based data center

As shown in Fig. 1, virtualization is the key feature for a cloud-based data center. Each application service is provided by a group of VM (Virtual Machine) nodes, which are identified belonging to the same tenant. Meanwhile, each VM node contains a certain virtualized hardware resources (CPU, Memory, and Disk, etc) on demand from resource servers. Servers are usually divided into two distinct types, compute (also called, computing) nodes and storage nodes [10]. Compute nodes host multiple VMs and provide such CPU and memory resources on demand, meanwhile multiple physical disk devices are organized and managed by storage nodes to provide I/O service. Resource pooling is the fundamental concept to achieve Service Level Agreement (*SLA*) of each tenant in cloud storage environments [4]. Storage Area Networks (SAN) becomes popular in clouds, because the network-based environment provides a scalable, reliable storage infrastructure to meet high-availability and elastic-scalability requirements [11]. Through the use of either iSCSI or Fibre Channel (FC) communication protocol, SAN accomplishes the ability that can interconnect any computer nodes with any storage nodes. Unfortunately, SAN faces an insurmountable obstacle in data transmission delay due to the network connectivity mechanism. This may be harmful for improving overall I/O performance [12].

3.2 The Benefits of the Cache-awareness Model in Clouds

As described in Equation 1, for a block reading request, C_1 denotes the average access latency in memory (usually in *ns* level), h_1 denotes the hit ratio in memory (tier-1 cache). C_2 denotes the average access latency in SSD (usually in μs level) and h_2 denotes the hit ratio in SSD (tier-2 cache). C_3 denotes the average access latency in HDDs (usually in *ms* level). Thus, the expected reading cost, *E(C)*, can be evaluated.

$$E(C) = C_1 \cdot h_1 + C_2 \cdot h_2 + C_3 \cdot (1 - h_1 - h_2). \quad (1)$$

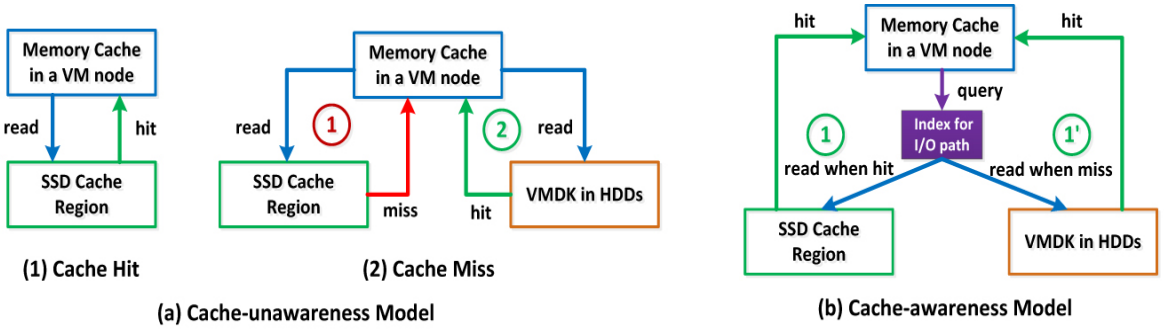

Fig. 2. The I/O path comparisons in the cache-unawareness model and the cache-awareness model

From Fig. 2, we see both C_2 and C_3 must include the network Round Time Trip (RTT) cost under the SAN circumstance (usually from μs to *ms*). For the cache-unawareness model, its C_2 is equal to RTT plus the block read latency from SSD, meanwhile its C_3 is composed of two parts: (1) Querying cost from SSD cache which is equal to RTT plus the query cost. (2) Reading cost from HDDs plus RTT. Therefore, a useless query to SSD cache through SAN links is inevitable even though the target block is not cached. Instead, for the cache-awareness model, its C_2 is equal to RTT plus the block read latency from SSD, while its C_3, is directly equal to RTT plus the block read latency from HDDs, bypasses the querying cost from SSD cache. Because the in-memory index query latency can be neglected. Compared to the cache-unawareness model, the main merit of the cache-awareness model is robustness to suit the property of the time-varying value of the hit ratio in SSD cache.

3.3 Dynamic Shared Cache Space Allocation

The purpose of dynamic cache space allocation is providing fair allocation and isolation to guarantee the *SLA* in clouds. Simply assuming that each tenant requires the same proportion of resources per partition leads to unfairness and inefficiency. Weighted max-min fair share schedule on the tenant basis or its variants are proposed for dynamic shared resource allocation. Given the shared cache capacity C, n tenants and the *i*th tenant's weight value w_i, where $1 \le i \le n$. The tenant i is guaranteed to obtain the cache capacity, $C_i = \frac{w_i}{\sum_{j=1}^{n} w_j} \cdot C$. The challenge here is how to measure w_i to reasonably partition the shared space to globally achieve high utilization. We found there are various multi-dimensional factors to influence w_i. We list and describe them but not limited as follows.

1) **Configured tenant priority:** VMs in different tenant execute differentiated tasks. Administer has the authority to configure the tenant priority according to the task importance.

2) **I/O access characteristics:** Different workloads show different access distributions, such as *Zipf*-like distribution in web, random distribution in database, etc. Meanwhile, the *read:write* ratio is varying change. A tenant service with a write-heavy workload may receive smaller benefit and lower performance due to the two inherent properties of SSD asymmetric read-write performance and write endurance.

3) **Time-varying I/O block request speed:** The I/O request spike may occurred in a period under two conditions: (1) dominant client I/0 requests in a tenant service. (2) low hit ratio in a tier-1 cache which incurs most I/O requests cannot be absorbed.

4) **Hit ratio:** Low hit ratio makes SSD cache inefficient. Thus, shifting the limited space for the tenants with low hit ratio to the tenants with high hit ratio can improve the cache utilization.

4 Implementation

4.1 Architecture

Fig. 3 depicts the high-level illustration of our proposed architecture. For each VM node, a corresponding SSD cache module is launched with tunable SSD space. Each module contains three components: (1) Cache monitor is used to record the interesting factors online as the form of I/O logs. The quota of each VM's cache space is dynamically tuned by the cache space allocator through analyzing the collected factors. (2) The purpose of Cache index is to decide whether the requested block can be satisfied in the SSD cache or not. The block request will be redirected to the SSD, once the index reports the target block is cached in SSD. Otherwise, the block request will be satisfied from the HDDs. Through I/O path identification, an extra and useless RTT for block query from VM to SSD can be avoided. (3) The LRU-based list is used to decide which cached block in SSD will be evicted as the victim to make free space for new block insertion when the cache space has been filled up. The design of the three components is described as follows.

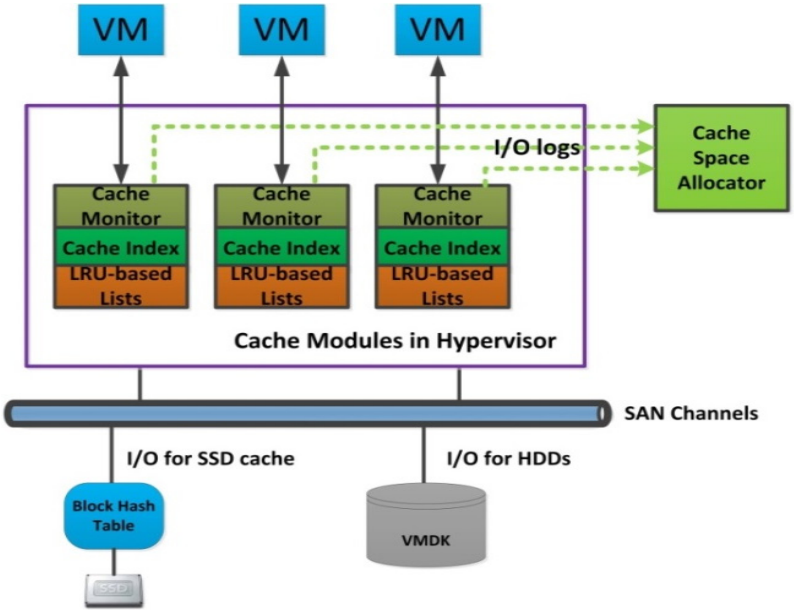

Fig. 3. An illustration of the proposed architecture when SSD as a shared tier-2 cache in clouds

(1) **Cache monitor:** the purpose is to record some mentioned factors online including tenant priority, *read:write* ratio, IOPS, total block reading times and total block reading hit times for evaluating hit ratio. Through the recorded log files, we can conduct the weight value of each tenant to optimize weighted max-min fair share schedule. As shown in Fig. 4, the measurements are usually based on the following fixed and discrete time-windows:

- *Measurement time-window (I)*: the interested factors are sampled and tracked during the each *I.* For instance, the recording line of the log file is written when the *I* is arrived. The size choice of a particular *I* highly depends on the desired responsiveness from the system. Once the system needs to swiftly react to the changes of each factor on a fine time-granularity, then a small value of *I*, such as in several seconds, should be chosen. On the other hand, if the system needs to accurately adapt to long term variations in the workload over a coarse time-granularity, then a large value of *I*, such as in minutes or hours, may be chosen. *I* is set to 1 minute in our default.

- *History (H)*: the history interval size consists of a group of finite measurement time-windows in sequence (*N* in Fig. 4). The monitoring module maintains this finite recent history values for each factors to predict the next cache size allocation. *H* is set to 30*I* (30 minutes) in our default.

- *Allocation Window (W)*: the recorded log lines from the past history are used to predict the cache size allocation for the next *M* measurement time-window units. Thus, the strategy of the cache size allocation is renewed in every *W* time units. *W* is set to 30*I* (30 minutes) in our default.

Notice that, the history and the allocation window are implemented as sliding windows.

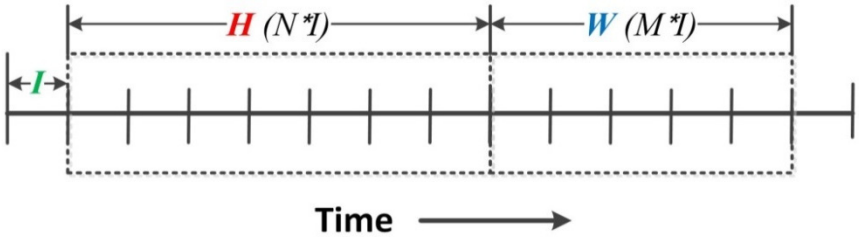

Fig. 4. Time windows are used for monitoring, prediction and allocation

(2) **Cache index:** we build the cache-awareness model as shown in Fig. 2(b). We propose our improved Bloom Filter (BF) as the index data structure to decide whether the block request can be read from SSD cache or not within a controllable and acceptable false positive probability (*fpp*) [1]. A BF is described by a bit array of m bits. k independent (pseudo) random hash function are used to map bit positions in the bit array for each block request. The memory overhead of a standard BF is very limited, each key of the indexed element only consumes about $1.44 \cdot log_2(fpp)$ bits in average. Take an example, if the expected fpp is initialized as about 0.78% (namely, $1/2^7$), the average bit consumption of a key is only about 10.08 bits (≈ 1.26 bytes). As shown in Fig. 5, suppose that the bit array size $\boldsymbol{m}$ of a BF is 16, the hash function number k is 3. First, block $\boldsymbol{x_1}$ is inserted to the array for indexing when $\boldsymbol{x_1}$ is cached in SSD. The mapping positions are 1, 6, and 10 by the three hash functions respectively (assuming the range of the array is [0, 15]). Then, block $\boldsymbol{x_2}$ is inserted to the array by the same way. The mapping positions are 6, 9, and 13 by these hash functions respectively. We can see the 7th bit position is shared by $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$. Suppose that there is a membership query for a block $\boldsymbol{y}$ whether it can be read from SSD cache or not. We see that the mapping positions of y are 1, 6, and 13. However, these three positions have been set to 1 by $\boldsymbol{x_1}$ and $\boldsymbol{x_2}$. Thus, false positive has been unexpected encountered that $\boldsymbol{y}$ is not existed in SSD cache but reports it is in. As a result, the block is finally redirected and read from the HDDs. Formally speaking, if the mapping positions of any element have been set by other elements, a false positive will be happened for its membership query. In Fig. 5, the actual fpp is about 33.3%.

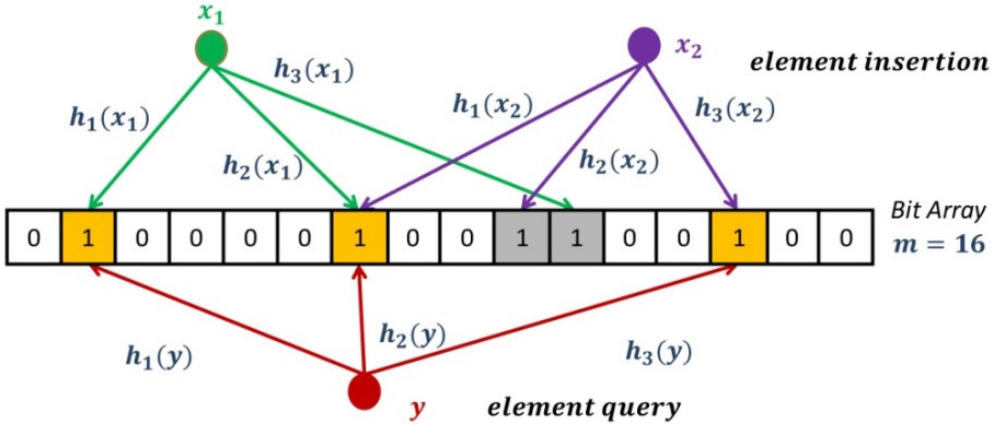

Fig. 5. An example of a standard BF

The space size of the standard BF must be predetermined based on the expected number of elements to be stored. However, we cannot predict the number of elements in a dynamic environment in advance. As shown in Fig. 6, suppose there are V disjointed and independent tenants $\{X_1, X_2, \ldots, X_V\}$. All the tenants are competing for a limited and shared pre-allocated SSD space for caching which can store C blocks at most. The major challenge is how much space should be preassigned to a tenant $X_i (1 \leq i \leq V)$. The space allocated to set X_i is denoted as C_i which must satisfy Equation 2. Consequently, the combination value of the formalized tuple with V items $< C_1, C_2, \ldots, C_V >$ is equal to $\binom{C+V-1}{C}$, which is deduced by the mathematical selection problem in *Combinatorics*. Therefore, the cache size allocated to a tenant X_i may change dynamically (either reduction or inflation). In order to support the property of dynamic size allocation on a tenant basis in clouds, we have designed a stand-

ard BF variant, called Par-BF [17], for supporting dynamic cache size partition. Par-BF is made up of a series of one or more sub-BFs. When the current sub-BF gets full, a new one is added to the list to accept new element insertions. Typically, the capacity of each sub-BF is much less than overall pre-allocated space, namely C, to meet different allocation granularities. The expected *fpp* is tuned in a very limited range (e.g. $\approx 0.39\%$, when k is set to 8). Moreover, the wrongly reported blocks due to the false positive will be corrected by SSD cache and read from HDDs without other extra processing burdens. The more interesting details can be found in [17].

$$\sum_i^V C_i = \mathrm{C}, \text{where } 0 \le C_i \le C. \quad (2)$$

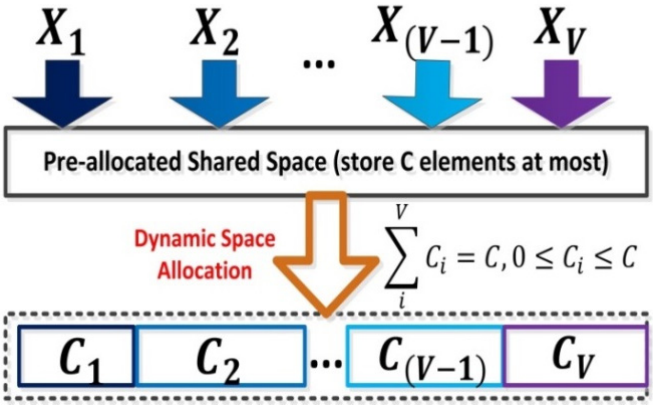

Fig. 6. The size allocated to each tenant X_i may change dynamically when V disjointed and independent tenants $\{X_1, X_2, \ldots, X_V\}$ to compete for a limited and shared pre-allocated SSD cache space

(3) **LRU-based lists:** we use ARC cache replacement algorithm [18] in default. It has been exhibited that single-level cache replacement policies perform very poorly when used in multilevel caches, because temporal localities of requests may be filtered out by the LRU-based policies in tier-1 cache [2]. In response to evolving and changing access patterns, ARC dynamically, adaptively, and continually balances between the recency and frequency components in an online and self-tuning fashion.

In addition, *Block Hash Table* is used for mapping the HDD-level logical block address into the SSD-level logical block address. The simple but effective address mapping policy is called as set associated hash where the Tier-2 cache is divided up into a number of fixed size sets (buckets) to do linear probing in a set to find blocks [6, 21].

4.2 Dynamic Space Allocation

Fig. 7 gives the hierarchy of dynamic cache space allocation according to the weighted max-min fair share schedule on a tenant basis. There are totally three different levels of space granularities. From the coarse-grained to the fine-grained, the whole shared SSD cache space is partitioned into several of independent sub-space. Each sub-space is managed by the corresponding tenant with fully considering the recorded factor values. Due to the dynamic property of either inflation or deflation in sub-cache space of a tenant, the whole SSD space is divided into a group of fixed-size units (we called it as *chunk*), which are considered as the smallest space for allocation and deallocation managements. Each VM node only requests its quota of the SSD space from its tenant through calculating how many chunks has been obtained. Through this organized division, we can greatly simplify the cache space management.

The weighted value of each tenant is conducted from the global optimization. Assuming there is n tenants, Equation 3 shows our solution to measure weight value w_{i} of the tenant i. We have listed four factors in the Section 2.2 including *tenant priority*, *read:write (r:w) ratio*, *read speed*, and *hit ratio*. The vector $\beta = [\beta_1, \beta_2, \beta_3, \beta_4]$ is denoted in sequence to measure the importance of each factor where $\sum_{j=1}^{4} \beta_j = 1$. α_{ij} is the normalized factor to represent the jth factor state of the tenant i. All the state values are recorded by the cache monitor. Take an example, α_{i4} denotes the normalized hit ratio state which is calculated by $\frac{hr_i}{\sum_{k=1}^{n} hr_k}$ where hr_i is recorded as the hit ratio of the tenant i. The other three normalized factor state can be also calculated using the same method.

$$w_i = \sum_{j=1}^{4} \alpha_{ij} \cdot \beta_j, where\ 1 \le i \le n. \quad (3)$$

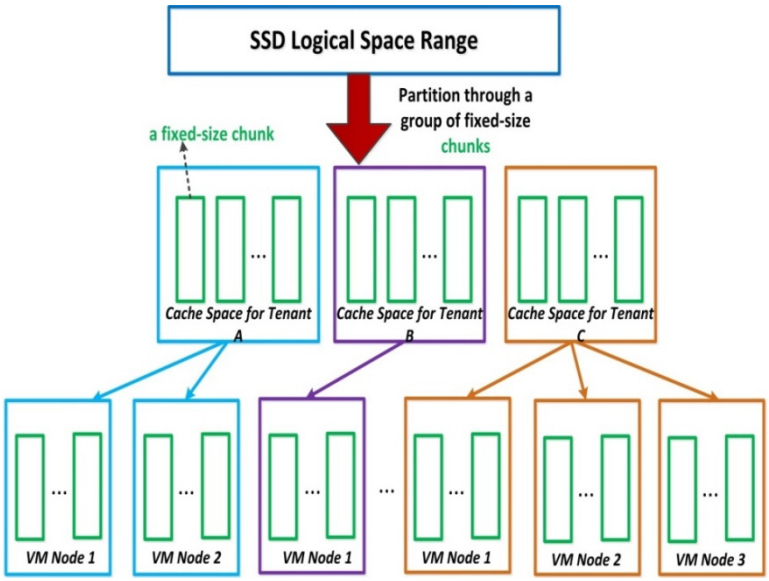

Fig. 7. Hierarchy of SSD cache space

5 Evaluation

Our evaluation mainly includes two testing targets. (1) Compared to the cache-unawareness model, how much latency can be reduced for a block reading in the cache-awareness model. (2) The reasonability of our mechanism to measure w_i of each tenant i in the weighted max-min fair share schedule.

5.1 Test Setup

Each Compute node has two Intel E5-2670 CPUs with GigE interface, 64GB of memory, and run Ubuntu 13.02 61-bit version. For storage nodes, our hardware testbed has a 2TB size 64K-stripe RAID-0 Flash arrays based on FusionIO Drive and 32TB size 64K-stripe RAID-5 HDDs. The Compute nodes and storage nodes are interconnected with each other through SAN. We have built a cloud orchestration prototype [29] based on *Openstack* to manage VM-nodes, especially making use of the standard component *Cinder* to manage a block device. Fig. 8 gives the top-to-down modules of the experiment platform. The data request behavior of each VM-node is simulated by the trace file, which is parsed to generate read requests for K-V items. The tier-1 memory cache is managed by the *Openstack Nova*, and the both block devices, SSD and HDD, are managed by *Cinder*.

On the VM-node side, a software simulator is developed to generate key-value (K-V) workloads following the principles of the Yahoo Cloud Storage Benchmark (YCSB) [28]. Two representative workloads: a Zipf-like ($\alpha = 0.99$) distribution and a random distribution are generated to validate the cache-awareness model, respectively. It has been found that the popularity of block requests for web is uneven [27] (following Zipf-like distribution), some of which are accessed many times, while others are visited in fewer times. In contrast, uniformly selecting a block for a request is the worst situation for caching, since the most-likely to be reused blocks cannot be identified. We use both distributions as the optimal- and the worst- bounds for hit ratio comparisons by testing different caching models. YCSB uses the Monte-Carlo method to generate typical access distribution. At first, a certain amount of unique K-V items are generated as the data sources. We use Berkeley DB as the K-V containers to store and organize data sources in the corresponding VMDK. Then, the K-V reading requests for those items are simulated in order by following a either random or Zipf-like probability distribution. Finally, each request is recorded as one line of the trace file in sequence. For simplicity, the size of each K-V item is set to 4KB. The size of data sources is 256GB, while the total data size of each workload for reading achieves to 16TB, namely having $16\text{TB}/4\text{KB} = 4 \times 10^9$ requests.

Meanwhile, we use the MSR Cambridge data-center trace [19] to test our mechanism of dynamic shared-cache allocation. These traces are the results of about one week service which covered thirteen different kinds of servers. A variety of server workloads can exhibit diverse access patterns, typically four server workloads: project directory (proj), web proxy (prxy), source control (src), and web SQL server (web). Thus, there are four VM-nodes, each of which simulates one of the four server workloads. Meanwhile, we have assumed that each corresponding VMDK has stored the sever data sources. The chunk size is initialized as 64MB.

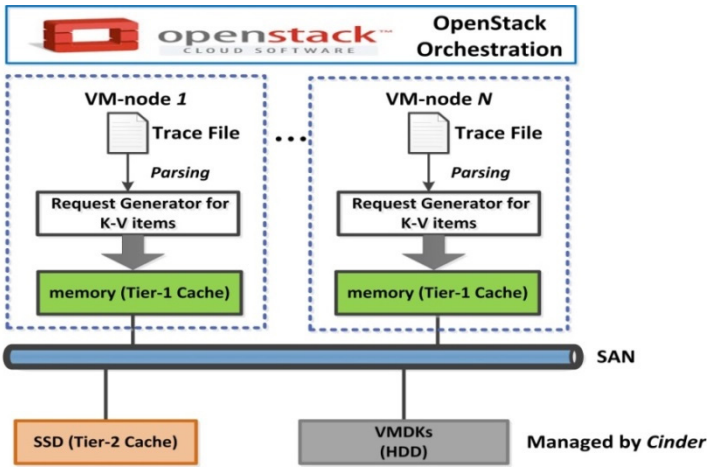

Fig. 8. Architecture of the experiment platform

5.2 The Benefit of the Cache-awareness Model

Three cases are used for performance comparisons including: (1) the model without SSD cache is considered as the baseline to validate whether using SSD cache can gain performance improvement or not. Thus, the block is directly read from *VMDK* when the block is missed in tier-1 cache. (2) the SSD cache with cache-unawareness model is considered to validate the significance of tier-2 cache index to avoid the useless RTT queries. (3) Our proposed SSD cache with the cache-awareness model.

Two VM-nodes are instantiated to simulate the above mentioned K-V workloads following Zipf-like and random distributions, respectively. The SSD cache size of each VM-node is set to 32GB. Fig. 9 shows the comparison results. We draw four conclusions as follows.

1. Cache-awareness performs better than the other two from about 1× to 4×. Zipf-like workload distribution makes high hit ratio in memory cache (> 70%) with the cache size from 32MB to 16GB. Whereas, the workload distribution following random access behavior makes the low hit ratio in memory cache (< 10%) with the cache size grows from 32MB to 16GB. The cache-awareness model is robust no matter status of the current hit ratio of the memory cache.

2. As shown in Fig. 9(a), the cache-unawareness model performs even worse than the case without the SSD cache. We analyzed two possible reasons: (1) the high hit ratio of memory cache absorbs a majority of block requests, as a result, the SSD cache plays a little role to improve the overall performance. (2) The costly RTT query time for SSD cache makes the cache-unawareness model ineffectively.

3. Tier-2 cache can improve performance more obviously only when the memory cache is in status of the lower hit ratio.

4. We validate that the memory overhead of BF-based index is limited, while the performance negative effect of *fpp* can be neglected. Only tens or hundreds of MBs is needed in this simulation according to the practical *fpp* tunings [17].

5.3 Measuring w_i of tenant i

We sample the four data-center traces and record the four factors in monitor module in the first day time. The statistics of the four recorded factors in four server workloads are given in Table 2. The matrix value $\boldsymbol{\alpha}$ is computed to normalize the four recorded factors of each server trace. The result of $\boldsymbol{\alpha}$ is shown in Equation 4, which is computed by the recorded values in Table 2. For instance, the vector $\alpha_1 = [\alpha_{11}, \alpha_{21}, \alpha_{31}, \alpha_{41}]^T$ is used to normalize the priority values of *proj*, *prxy, src*, and *web*, such as $\alpha_{11} = \frac{8}{8+10+6+8} = 0.25$ represents the normalized priority value of *proj*. The other three factors of each sever trace are normalized by the same. β is initialized as [0.2,0.2,0.2,0.4] to indicates the importance degree of each factor. Therefore, hit ratio is more important than the other three twice, while the importance of the remaining three factors is same. Thus, the vector w is computed as [0.15,0.54,0.18,0.13] through Equation 3, such as w_1 is equal to $(0.2 \cdot 0.25 + 0.2 \cdot 0.14 + 0.2 \cdot 0.29 + 0.4 \cdot 0.035 = 0.15)$. According to the weighted max-min fare share schedule, the tenant i is guaranteed to obtain at least $\boldsymbol{C_i} = \frac{w_i}{\sum_{j=1}^{4} w_j} \cdot \boldsymbol{C}$ cache space where C denotes the whole space of SSD cache (2TB in our test setup). The final space allocation result is: *proj* gets 0.30TB, *prxy* gets 1.08TB, *src* gets 0.36TB, and *web* gets 0.26TB.

$$\boldsymbol{\alpha} = \begin{bmatrix} 0.25 & 0.14 & 0.29 & 0.035 \\ 0.32 & 0.70 & 0.49 & 0.60 \\ 0.18 & 0.10 & 0.037 & 0.28 \\ 0.25 & 0.06 & 0.19 & 0.085 \end{bmatrix} \quad (4)$$

Table 2. The statistics of four recorded factors in four server workloads (The memory cache size is 4GB)

Trace	priority (α_1)	*r:w* ratio (α_2)	read speed (α_3)	hit ratio (α_4)
proj (T_1)	8	3.25:1	164/s	4.82%
prxy (T_2)	10	16.26:1	278/s	82.6%
src (T_3)	6	2.34:1	21/s	38.49%
web (T_4)	8	1.42:1	106/s	11.60%

Fig. 10 validates our mechanism to measure w_i in each tenant i. We can see our space allocation method outperforms the other three. The overall hit ratio of our method achieves to 42.8%, while the other three values are in the range of 20% to 30%. Hence, the SSD space allocation measured by our weighted fair share schedule achieves higher utilization.

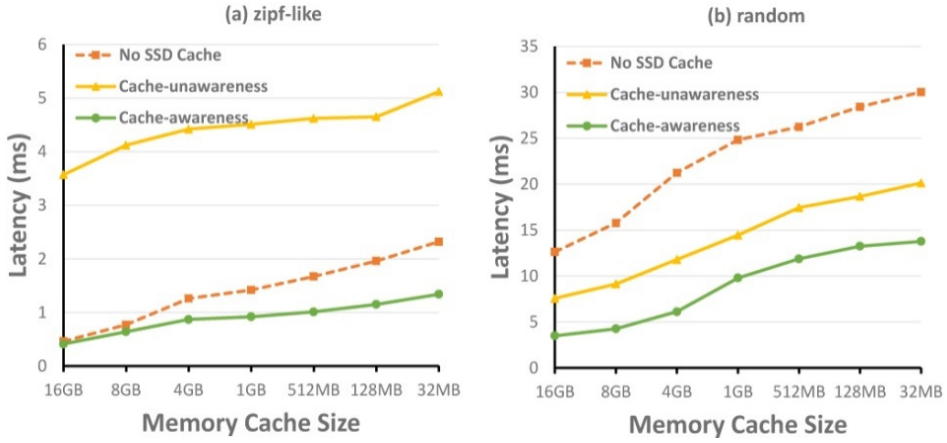

Fig. 9. The results of the average K-V item reading latency of the three cases

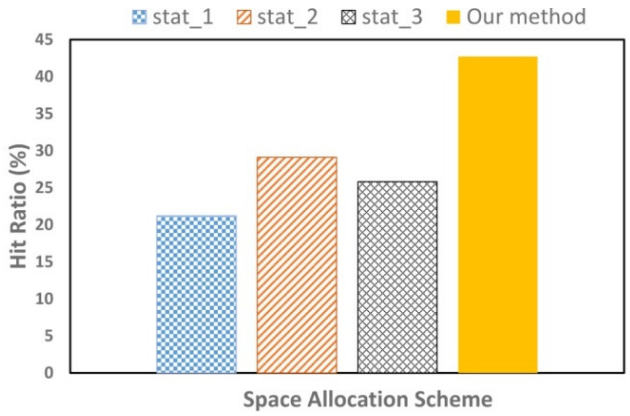

Fig. 10. The results of overall hit ratio in four weight values for weighed max-min fair scheme. The "stat_1" is the uniform allocation, its *w* is [0.5*T*, 0.5*T*, 0.5*T*, 0.5*T*]. The *w* of "stat 2" is *[0.5T, 0.625T, 0.325T, 0.5T]* only according to configured tenant priority (α_1). The *w* of "stat_3" is empirical value that is equal to *[0.4T, 0.6T, 0.6T, 0.4T]*

6 Conclusion and Future work

This paper mainly validates the our improvements when designing SSD as a shared cloud cache: (i) the robustness of the cache-awareness model without any assumption in the hit ratio state of the memory cache, (ii) four multi-dimensional factors are fully explored, which may greatly affect the quota assignments of the shared cache space.

In the future, we intend to measure more factors which further affect the dynamic space allocation. We plan to validate our improvements through various related experiments in a production system. This paper lacks the further discussion in strategy of multiple resource allocation, namely how the SSD space allocation strategy influences the overall performance of a tenant. Therefore, the study of integrating SSD as a cloud cache for considering multi-resource fairness has a promising future.

Acknowledgement

We authors would like to thank anonymous reviewers for the valuable feedback to improve this paper. This work is partially supported by the following NSF awards: 1439622, 1305237, 1421913, 1217569 and 1115471. The work of Huang is supported by the Joint Program of National Science Foundation of China-Guangdong under grant No.U1301256.

References

[1] B.H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, Vol. 13, pp. 422-426, 1970.

[2] D.L. Willick, D.L. Eager, R.B. Bunt, "Disk cache replacement policies for network fileservers," in *Proceeding of the 13th ICDCS*, pp. 2-11, 1993.

[3] J. Gantz, D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the Far East," *IDC iView: IDC Analyze the Future*, 2012.

[4] M. Armbrust et al., "A view of cloud computing," *Communications of the ACM*, Vol. 53, No. 4, pp. 50-58, 2010.

[5] D. G. Andersen and S. Swanson, "Rethinking flash in the data center," *IEEE Micro Magazine*, Vol. 30, No. 4, pp. 52-54, 2010.

[6] Y. Liu, X. Ge, X. Huang, D.H. Du, "MOLAR: A cost-efficient, high-performance hybrid storage cache," in *Proceeding of the IEEE CLUSTER'13*, pp. 1-5, 2013.

[7] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, M. Storer, "Mercury: Host-side flash caching for the data center," in *Proceeding of the IEEE 28th Symposium on Mass Storage Systems and Technologies*, pp. 1-12, 2012.

[8] J. Hwang, A.J. Uppal, T. Wood, H.H. Huang, "Mortar: filling the gaps in data center memory," in *Proceeding of the 4th annual Symposium on Cloud Computing*, Article No. 30, 2013.

[9] R. Koller, L. Marmol, R. Rangaswami, S. Sundararaman, N. Talagala, M. Zhao, "Write policies for host-side flash caches," in *Proceeding of the 11th USENIX Conference on File and Storage Technologies (FAST'13)*, pp. 45-58, 2013.

[10] Q. Zhang, L. Cheng, R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, Vol. 1, No. 1, pp. 7-18, 2010.

[11] S. Milanovic, N.E. Mastorakis, "Internetworking the storage area networks," *WSEAS Transactions on Communications*, Article No.1, pp. 8-13, 2002.

[12] A. Gulati, C. Kumar, I. Ahmad, "Storage workload characterization and consolidation in virtualized environments," in *Proceeding of the Workshop on Virtualization Performance: Analysis, Characterization, and Tools (VPACT)*, 2009.

[13] B.S. Gill, "On multi-level exclusive caching: Offline optimality and why promotions are better than demotions," in *Proceeding of the 6th USENIX Conference on File and Storage Technologies*, Article No. 4, 2008.

[14] T.M. Wong, J. Wilkes, "My cache or yours? Making storage more exclusive," in *Proceeding of the USENIX Annual Technical Conference*, pp. 161-175, 2002.

[15] G. Hinton, D. Sager, M. Upton, D. Boggs, et al., "The microarchitecture of the Pentium 4 processor," *Intel Technology Journal*, 2001.

[16] D. Shue, M.J. Freedman, A. Shaikh, "Performance isolation and fairness for multi-tenant cloud storage," in *Proceeding of the 10th USENIX Conference on Operating Systems Design and Implementation*, pp. 349-362, 2012.

[17] Y. Liu, X. Ge, D.H. Du, X. Huang, "Par-BF: a parallel partitioned Bloom filter for dynamic data sets," in *Proceeding of the 2014 DISCS workshop in conjunction with SC'14*, pp. 1-8, 2014.

[18] N. Megiddo, D.S. Modha, "ARC: A Self-Tuning, Low Overhead Code Positioning Replacement Cache," in *Proceeding of the Second USENIX Conference File and Storage Technologies (FAST)*, pp. 115-130, 2003.

[19] D. Narayanan, A. Donnelly, A. Rowstron, "Write off-loading: Practical power management for enterprise storage," *ACM Transactions on Storage (TOS)*, Vol. 4, No.10, 2008.

[20] A. Leventhal, "Flash storage memory," *Communications of the ACM*, Vol. 51, pp. 47-51, 2008.

[21] Flashcache, https://github.com/facebook/flashcache/.

[22] Dm-cache, http://visa.cs.fiu.edu/dmcache.

[23] F. Meng et al., "vCache Share: Automated Server Flash Cache Space Management in a Virtualization Environment," in *Proceeding of the USENIX ATC'14*, pp. 133-144, 2014.

[24] C. Albrecht, A. Merchant, A.M. Stokely, et.al, "Janus: Optimal flash provisioning for cloud storage workloads," in *Proceeding of the USENIX ATC'13*, pp. 91-102, 2013.

[25] D. Narayanan et al., "Migrating server storage to SSDs: analysis of tradeoffs," in *Proceeding of the 4th European conference on Computer systems (EuroSys'09)*, pp. 145-158, 2009.

[26] R. Appuswamy, D.C.van Moolenbroek, and A.S. Tanenbaum, "Cache, cache everywhere, flushing all hits down the sink: On exclusivity in multilevel, hybrid caches," in *Proceeding of the IEEE MSST'13*, pp. 1-14, 2013.

[27] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proceeding of the IEEE INFOCOM'99*, Vol. 1, pp. 126-134, 1999.

[28] B.F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM symposium on Cloud computing*, pp. 143-154, 2010.

[29] X. Ge et al., "OpenANFV: accelerating network function virtualization with a consolidated framework in Openstack," in *Proceedings of the SIGCOMM'14*, pp. 353-354, 2014.

[30] Openstack, http://www.openstack.org/.

[31] A. Ghodsi et al., "Dominant resource fairness: Fair allocation of multiple resource types," in *Proceeding of the NSDI'11*, pp. 323-336, 2011.

[32] A. Demers, S. Keshav, S. Shenker, "Analysis and simulation of a fair queueing algorithm," *ACM SIGCOMM Computer Communication Review*, Vol. 19, pp. 1-12, 1989.

[33] A.K. Parekh, R.G. Gallager, "A generalized processor sharing approach to flow control in integrated services networks: the single-node case," *IEEE/ACM Transactions on Networking (ToN)*, Vol. 1, No. 3, pp. 344-357, 1993.