Hierarchical Consistent Hashing for Heterogeneous Object-based Storage

Jiang Zhou, Wei Xie, Qiang Gu, Yong Chen
Department of Computer Science
Texas Tech University, Lubbock, TX, USA
{jiang.zhou, wei.xie, qiang.gu, yong.chen}@ttu.edu

***Abstract*—Distributed storage systems play an increasingly critical role in data centers to meet the ever-increasing data growth demand. Heterogeneous storage systems, with the co-existence of hard disk drives (HDDs) and solid state drives (SSDs), can be an attractive distributed store solution due to the balanced performance, large capacity, and economic cost. The consistent hashing distribution algorithm that is widely used in distributed storage systems can achieve scalable and flexible mappings, but do not take full advantages of different characteristics of heterogeneous devices. In this research, we propose a hierarchical consistent hashing (HiCH) algorithm to better manage data distribution in a heterogeneous object-based storage system and better explore the potential of heterogeneous devices. HiCH divides heterogeneous storage devices into different buckets and applies separate consistent hashing rings for each bucket. It places data into various hashing rings according to the hotness, access time, and other data access patterns. The evaluation based on the Sheepdog, a distributed object-based storage system, confirms that HiCH can improve the performance of storage systems, and also make better utilization of heterogeneous storage devices.**

I. Introduction

Cloud storage has been a popular solution for enterprise and end users to store massive amounts of data without owning the actual hardware. A cloud storage infrastructure relies on the storage system that distributes data across a large number of servers. Such a distributed storage system plays an increasingly critical role in data centers to meet the ever-increasing data growth. The most critical element of a distributed storage system is the data placement component, which is responsible for partitioning and placing data across servers in the data center. Data placement can dramatically affect the performance of a distributed storage system. It also has to be able to distribute the workload across the data center in a balanced way to prevent hot spots. Many distributed storage systems are designed to have metadata servers to dedicate for handling data placement. However, the existence of metadata servers limits the scalability of the data centers. Decentralized storage systems that rely on distributed hash tables avoid the use of metadata servers and can achieve better scalability compared to distributed storage systems with metadata servers.

In the meantime, object-based storage is a rapidly growing storage architecture in recent years that provides high performance data access [1], [2]. Different from commonly used storage, including direct-attached storage (DAS), network-attached storage (NAS), and storage area networks (SANs), object-based storage represents its storage as objects instead of the traditional block or file API. A storage object is a logical collection of bytes with methods for access, attributes for describing characteristics of the data, and security policies that prevent unauthorized accesses. It has variable size and can store various data structures, such as metadata, data files, blocks, or data tables. In a typical object-based storage, it usually decouples data and metadata management in which the objects are stored in object-based storage devices (OSDs). The client can directly access the objects in OSDs, thereby avoiding complicated data path and improving the I/O performance. In general, object-based storage enables better allocation and provision of resources in storage devices and has great potential to effectively manage massive amounts of data.

The object storage and hashing-based distribution [3], [4] design are usually adopted to address scalability and availability problems. The object storage uses a flat structure, instead of a hierarchical structure in traditional file systems. This key-value style storage system fits well with massive data management and provides symmetrical architecture for object distribution and replication. The hashing-based management of data-node distribution eliminates the need of a centralized table for mapping data to nodes. It uses algorithms, such as consistent hashing [5] or a pseudo-random number method [6], [7], [8], [9], to determine the data placement on the storage nodes. The benefit of it, compared to mapping tables in metadata servers, includes better performance, high scalability, easy management, and inherent data replication capability. Existing storage systems using consistent hashing-based distribution have gained increasing attention, such as the Dynamo [3], Chord [10], Cassandra [11], Ceph [12], and Sheepdog [13].

The hashing-based algorithms can achieve efficient data placement, However, they can not provide balanced data distribution in a *heterogeneous* storage system environment. The performance of storage devices is continually improved with the emergence of storage class memory (SCM), such as solid state drives (SSD) and phase change memory (PCM) [14]. The storage system can benefit from such devices that provide high bandwidth, low latency, and mechanical-component-free, in addition to conventional hard disk drives (HDD). A tiered storage design with heterogeneous devices renders an ideal solution due to the cost-effective merits of HDDs and SCMs.

The general design goal of a heterogeneous storage system involves making the best use of fast storage devices such as solid state drives, while at the same time exploiting the massive capacity of hard disk drives.

In this paper, we present a novel *hierarchical consistent hashing algorithm (HiCH)* designed specifically for data placement on heterogeneous storage systems. The HiCH algorithm divides heterogeneous devices into different buckets and applies separate consistent hashing rings for each bucket. It has a hierarchical structure, in which the data is placed on one or more hashing rings with the consistent hashing algorithm. A data replacement strategy is used to transfer data among hashing rings. Evaluations show that HiCH can improve the performance of storage systems, and also make better utilization of heterogeneous storage devices.

The rest of this paper is organized as follows. Section II introduces the proposed HiCh algorithm, including its model, data distribution algorithm, object read/write operation, as well as replacement strategy. Section III presents the evaluation results based on the Sheepdog system evaluation. Section IV reviews existing studies and compares with this study. We summarize this research study in Section V.

II. The HiCH Algorithm

A. Hierarchical Consistent Hashing Model

The consistent hashing algorithm is widely used in modern distributed storage systems [3], [10], [11], [13] due to its simplicity and scalability. It uses the hashing functions to map both the data and storage nodes to a hash ring (a certain range of hash values), and places data onto those storage nodes. It provides a key value store and uses hashing functions to assign data to nodes. In addition, it is designed with little data migration required when a storage node addition or removal occurs [5].

When hashing a storage node to only one location on the hash ring, it is very difficult to achieve a uniform data distribution. To achieve better uniform distribution, consistent hashing uses *virtual nodes* by copying the storage node to multiple locations on the ring. A proportional distribution can be achieved by manipulating the number of virtual nodes for different storage nodes according to the node capacity. It achieves load balance as each storage node's capacity is fully used at the rate proportional to its capacity. Numerous variants and methods based on consistent hashing have also been proposed to alleviate the characteristic difference among heterogeneous nodes [15], [6], [16]. However, they only focus on one characteristic of nodes (e.g., device capacity), but ignore other dimensions (e.g., throughput) that are also important for data placement.

The proposed hierarchical consistent hashing algorithm (HiCH) is based on the consistent hashing algorithm, which not only keeps the consistent hashing's inherent features but is also designed to work in heterogeneous environments much more efficiently. The HiCH algorithm divides heterogeneous nodes into different buckets and maintains a consistent hashing ring for each bucket. The hashing rings of all buckets construct a *hierarchical consistent hashing*, where the nodes in higher level bucket have better throughput. It forms a multi-tier storage system, in which the data is placed on consistent hashing rings at different levels. Figure 1 shows the model of the hierarchical consistent hashing in a hybrid storage system with HDD and SSD nodes. The heterogeneous nodes are divided into two buckets and construct two hashing rings: an HDD hashing ring and an SSD hashing ring. In each hashing ring, the HiCH uses the consistent hashing algorithm to map data on storage nodes. Virtual nodes can also be used in the hashing ring to achieve better load balance. From this illustrative example shown in Figure 1, it can be seen that the object x is placed on node A in the SSD hashing ring and nodes E and F in the HDD hashing ring with three replicated copies (the replica number is 3).

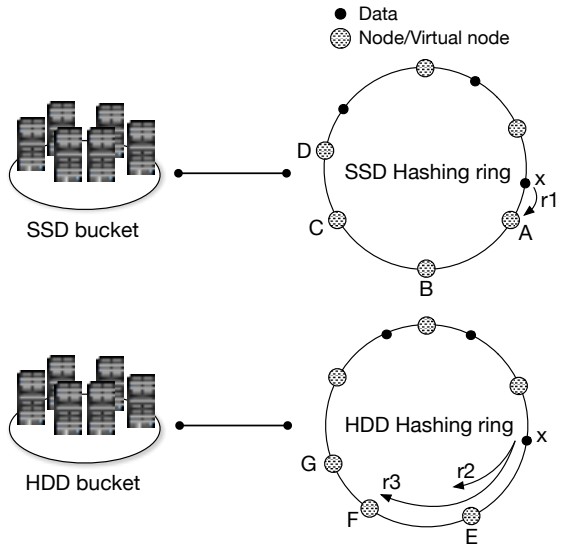


Fig. 1: Hierarchical consistent hashing model

B. Data Distribution

The HiCH algorithm distributes data among heterogeneous nodes for data placement. Figure 2 shows a diagram of the HiCH algorithm in a heterogeneous storage system with HDD and SSD nodes. There are two hashing rings for HDD and SSD bucket, respectively. When new data objects come, the first choice is to place the data on the SSD hashing ring. The HiCH selects the appropriate SSD node for data placement with the consistent hashing algorithm. Thus the SSD hashing ring is regarded as the cache for the HDD hashing ring. As the performance of SSDs is much higher than that of HDDs, the HiCH can improve the I/O throughput of the storage system when accessing data on the SSD hashing ring.

To enhance data availability and reliability, data replication is widely used in a distributed storage system. With replication, the data can be retrieved from other replicas if one copy is not accessible or corrupted. The HiCH algorithm makes replications by placing replicas on the hierarchical consistent hashing rings. Supposing there are three replicas, as shown in Figure 2, the HiCH algorithm places the first copy on the SSD

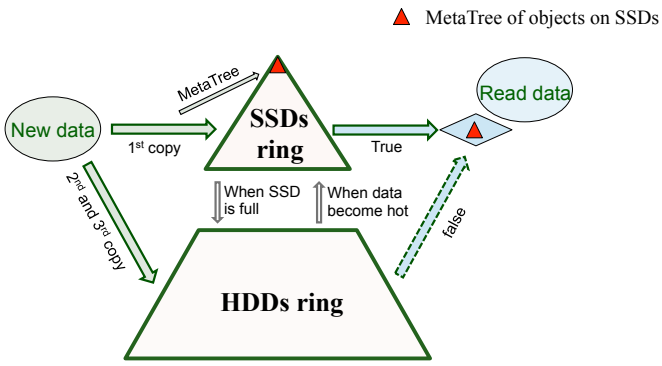


Fig. 2: Illustration of HiCH algorithm

hashing ring. The other two replicas are placed on the HDD hashing ring. For each hashing ring, the replica is mapped to the node according to the consistent hashing algorithm. With the increase of the amount of data being stored, the SSD hashing ring may have no sufficient capacity to store the data. Data movement from SSD to HDD hashing ring will happen at this time. On the other hand, the data on the HDD hashing ring may move to SSD hashing ring when the data is frequently accessed. The data replacement mechanism will be discussed in Section II(E). It is designed to combine the performance of SSDs with the capacity and economic efficiency of HDDs.

To achieve an efficient data movement, the HiCH algorithm maintains the data access information for each SSD node, called *metaTree*, for all data objects stored on the node. The data access information reflects the application pattern of I/O operations [17], [18]. The metaTree is organized like an AVL tree [19], in which the metadata of each object is stored as a node in it. The information for each object contains object ID, hotness count, last access time, and a boolean flag to indicate the accessibility of the object (as seen in the data structure of *MetaTreeNode*). The nodes in the metaTree are automatically sorted (based on the *object ID* by default) and it is efficient to look up, add or delete a node. In an object-based storage system, the metaTree can be persistently stored on the SSD bucket as objects.

```
struct MetaTreeNode
{
string   objectID;
int   hotnessCount;
int   lastVisitTime;
bool   accessible;
};
```

The HiCH will update the metaTree when placing an object on the SSD bucket. It adds a node in the metaTree for writing a new object. Other replicas will be placed on the HDD bucket. As numerous protocols [20] can be used to keep data consistent for replicas, it is not the focus of this study to address this issue in HiCH. Algorithm 1 describes the HiCH data placement algorithm.

Algorithm 1 HiCH Data Placement

INPUT: object ID x, replica number r;

1: $\triangleright place\ the\ 1st\ copy\ on\ SSD\ hashing\ ring$
2: $place(x,\ SSD,\ 1)$
3: $metaTree \rightarrow insert(new\ metaTreeNode(x,$
4: $hotnessCount = 0,\ current\ time,\ accessible = 1))$
5: **for** $i = 2;\ i \leq r;\ i++$ **do**
6: $\quad \triangleright place\ other\ copies\ on\ HDD\ hashing\ ring$
7: $\quad place(x,\ HDD,\ i)$
8: **end for**

C. Object Read Operation

In an ideal situation, each object would have one replica on SSDs and each read operation could be able to find the object on the SSD hashing ring. In practice, however, only a portion of objects are stored on SSD nodes due to the limited capacity. In the HiCH algorithm, the first step for the read operation is to search the metaTree and check whether the target object is located in the SSD bucket. If there is a *metaTreeNode* entry in the metaTree matching the target $objectID$, the HiCH will read data from the SSD bucket based on the consistent hashing algorithm. Otherwise, the object is not stored on the SSD hashing ring and should be retrieved from the HDDs. The *metaTree* not only provides the information about whether the target object exists on the SSD hashing ring, but also records the access pattern of the object with the hotness counter and last access time. After reading an object on the SSD hashing ring, HiCH updates the corresponding information of the *metaTreeNode* entry in the *metaTree*. The hotness counter of an object increases for each read operation, which means the object becomes hot when it is frequently read. The HiCH records such read pattern for each object on the SSDs in which the information can be used for data replacement.

The HiCH algorithm serves read operations from the SSD hashing ring whenever possible. However, miss still happens as the SSDs can only contain a portion of the objects, especially when the data access pattern changes. Note that even with data replication, all replicas of an object may be placed on the HDDs after data movement. At this time, HiCH will read the object from the HDD hashing ring if the object is not available in SSDs. The current read operation represents the data workload for the storage system and it is highly possible that the object will be read again. Then HiCH moves the recent read object from HDD hashing ring to SSD hashing ring. It places the object as new data and insert a *metaTreeNode* entry in the *metaTree*. After reading the object, HiCH updates *metaTree* for the object's hotness counter and access time. This update will enable new read operations, if there are any, access the target object available on the SSD hashing ring instead of HDDs. The algorithm for reading objects in HiCH is described in Algorithm 2.

D. Object Write Operation

The write optimization is often more complicated in storage systems due to I/O scheduling policy, data consistency, and other factors. The effectiveness is also not evident sometimes

Algorithm 2 Read object

INPUT: target object ID x;

1: $Struct * MetaTreeNode\ pr = metaTree \rightarrow find(x)$
2: **if** $pr == NULL$ **then**
3: $\quad read(x, HDD)$
4: $\quad move(x, SSD)$
5: $\quad metaTree \rightarrow insert(new\ metaTreeNode(x,$
6: $hotnessCount = 1,\ current\ time,\ accessible = 1))$
7: $\quad return$
8: **end if**
9: $read(x, SSD)$
10: $pr \rightarrow hotnessCount++$
11: $pr \rightarrow lastVisitTime = current\ time$

due to the existence of write-buffer and asynchronous writes. Additionally, applications often have the *write-once, read-many* [21] pattern. To better support the read performance, HiCH does not modify the *metaTree* for write operations. For each existing object, HiCH simply tries to find the object in the SSD hashing ring and performs a write on it. If the object is unavailable, HiCH accesses it from the HDDs with the consistent hashing algorithm. When the object has replicas, HiCH updates all copies for the object. Algorithm 3 describes the write operation in the HiCH algorithm.

Algorithm 3 Write object

INPUT: target object ID x;

1: $Struct * MetaTreeNode\ pr = metaTree \rightarrow find(x)$
2: **if** $pr == NULL$ **then**
3: $\quad write(x, HDD)$
4: $\quad \triangleright write\ other\ replicas\ on\ HDD\ hashing\ ring$
5: $\quad return$
6: **end if**
7: $write(x, SSD)$
8: $\triangleright write\ other\ replicas\ on\ HDD\ hashing\ ring$

E. Object Replacement Strategy

Compared to the large capacity of HDDs in modern distributed storage systems, the total capacity volume of SSD devices is relatively small. For each read, the object moves from the HDD hashing ring to the SSD hashing ring. Thus the data on SSDs will reach their space limit. On the other hand, some data in SSDs may not be frequently read after the first access. These data occupy the precious capacity of SSDs, which will affect the performance of the storage system.

The HiCH algorithm uses a data replacement strategy to move objects from SSDs to HDDs. It tends to achieve load balance between heterogeneous devices and improve the I/O performance by placing the hot data on SSDs. The replacement strategy will replace the infrequently read, least accessed objects with the newly accessed ones. To reduce the impact on I/O performance, HiCH is designed to trigger data replacement at a watermark when there is no sufficient space on SSDs, e.g., 90% full of the total capacity of SSDs. It moves data in batch from the SSD hashing ring to HDD hashing ring. During the movement process, HiCH modifies the *metaTree* on SSDs to reflect the migration of objects. It deletes the *MetaTreeNode* for each object that will be evicted. A significant benefit of considering the capacity for data replacement is to reduce the frequency of data movement and allow enough objects to be placed in the SSDs after each placement. However, the data movement should stop when there is enough free space on the SSDs. Otherwise, it would compromise the performance of the overall storage system since much of SSD capacity remains unused, which conflicts with our design principle discussed before.

The HiCH algorithm uses two watermarks, W_h and W_l, to decide the start and finish time of data replacement. The W_h and W_l are upper and lower proportion of objects occupying on the SSD bucket's total capacity. For example, the data replacement occurs at the condition of $W_h = 90\%$, and can stop at the condition of $W_l = 60\%$. The watermarks are adjustable in a real storage system depending on the configuration between SSDs and HDDs nodes. Another issue needs to be addressed is that the objects that are removed from SSDs need to be re-replicated to keep a fixed replica number in the entire system. The re-replication can be easily achieved using the consistent hashing algorithm. For example, assuming the replica number is 3, HiCH will select the *third* successor node on the SSD hashing ring for the data as the replica placement if there are already two replicas on it.

The HiCH algorithm moves data from SSDs to HDDs and places newly accessed data on SSDs. When the data replacement happens, it needs to decide which objects should be removed from the SSD hashing ring. Numerous replacement algorithms have been designed and most of them can be efficiently used for data replacement [22]. HiCH combines the features of LFU (least frequently used) and LRU (least recently used) algorithms for object replacement. As mentioned above, HiCH uses the *metaTree* to record the the hotness count and last access time for each object on SSDs. When replacing data, HiCH sorts the objects by their hotness counter. It selects a percentage (the value is $W_h - W_l$) of objects whose hotness counts are in the lower ranking. From these objects with lower hotness counter, HiCH excludes the objects that are recently accessed. If the time interval between the current time and last access time is less than a threshold T_{lag}, the object can be regarded as hot data and will not be removed from the SSDs. After data movement, the hot data are still resident on the SSD hashing ring while the latter also has sufficient space to store new objects.

The processing of data replacement, including data movement and re-replication, can be performed in an asynchronous way. As HiCH will search the object on HDD hashing ring if it is not on the SSD hashing ring, it will not compromise the data consistence and the entire system performance. Algorithm 4 describes the HiCH data replacement algorithm.

F. Data Migration

In a large-scale storage system, the cluster scale may change due to node addition or removal. It will affect the data distribution on the heterogeneous nodes. An ideal data

Algorithm 4 Object replacement algorithm

INPUT: high watermark W_h, low watermark W_l, time threshold T_{lag};

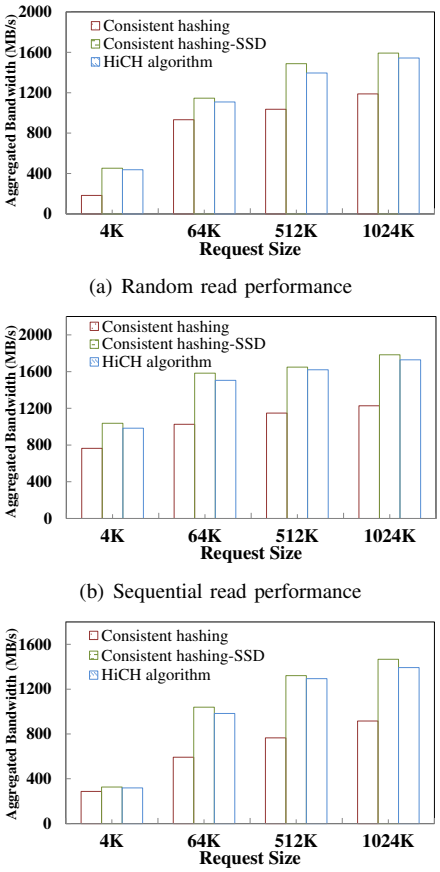
if $\frac{object\ amount}{total\ capacity\ of\ SSDs} > W_h$ **then**
  $\triangleright sort\ objects\ in\ metaTree\ by\ hotnessCount\ in\ an\ ascending\ order$
  $sort(metaTree)$
  while $\frac{object\ amount}{capacity\ of\ SSDs} > W_l$ **do**
    $\triangleright get\ an\ object\ in\ order\ from\ the\ sorted\ objects$
    $Struct * MetaTreeNode\ pr = getNext()$
    if $current\ time - (pr \rightarrow lastVisitTime) < T_{lag}$ **then**
      $continue$
    end if
    $remove(pr \rightarrow objectID, SSD)$
    $metaTree \rightarrow delete(pr \rightarrow objectID)$
    $re-replicate(pr \rightarrow objectID, HDD)$
  end while
end if

distribution should minimize the data movement among nodes while keeping load balance. Our HiCH algorithm is based on consistent hashing, which keeps the merits of it and can reduce the data migration when nodes membership changes. For node addition, HiCH achieves data movement in each hashing ring according to the conventional consistent hashing algorithm. For example, if an SSD node is newly added, HiCH moves objects among nodes based on the SSD hashing ring. Supposing the original system has N SSD nodes, HiCH will move $\frac{1}{N+1}$ amount of data from others to the new node. A similar movement occurs on the HDD hashing ring when adding an HDD node.

When a node removal happens, HiCH will re-replicate data for the objects on the removed node. As the HDD hashing ring often has multiple replicas for an object, HiCH can simply make a new copy for the object from existing objects in other nodes. Compared to HDD hashing ring, the situation of removing an SSD node is more complicated. The SSD nodes in HiCH can often have high utilization and removing a node would significantly affect the data access. The objects in the removed SSD node cannot be retrieved within the SSD hashing ring. HiCH addresses the issue by making replicas in the HDD hashing ring if the system needs to maintain a fixed number of object copies. An error-handling mechanism is triggered if an object is unavailable due to the removal of a node but still has a *MetaTreeNode* entry in the *metaTree*. HiCH will mark the *accessible* field in the object's *MetaTreeNode* from 1 to 0 at this time. When the object in the removed node is accessed, HiCH moves it from HDD hashing ring to SSD hashing ring and sets the *accessible* field back to 1 again. HiCH can seamlessly adapt to node addition/removal with minor overhead incurred.

III. Evaluation

In this section, we present the evaluation results of the proposed HiCH algorithm. We implemented the HiCH algorithm based on the Sheepdog [13], a distributed object-based storage system for virtual machine (VM) storage in cloud computing. The original Sheepdog storage cluster is the *baseline* system, which uses a consistent hashing-based data distribution policy to distribute objects on storage nodes. We use typical benchmarks to generate workload of data access for comparison in different scenarios.



(a) Random read performance

(b) Sequential read performance

(c) Sequential write performance

Fig. 3: FIO performance of different algorithms

We conducted the experiments on a local cluster with 30 storage nodes. In the storage nodes, half of them are equipped with dual 2.6 GHz Xeon 8-core processors, 128GB memory, and a 200GB Intel SSD (SSDSC2BA200G3T); others have dual 2.5 GHz Xeon 8-core processors, 32GB memory and a 500GB Seagate SATA HDD (ST9500620NS). Each storage node can be used as a client node for a VM instance, which was emulated by KVM-QEMU and configured with 2 vCPUs and 4GB RAM. HiCH constructs two hashing rings for HDD and SSD nodes, respectively. Additionally, the consistent hashing function we used is FNV-1 hash function [13] which is used by both Voldemort [23] and Sheepdog [13]. Sheepdog storage cluster was formed with a default replication factor of 3 and block size of 64MB.

A. I/O Performance

We first evaluate the I/O performance of different algorithms with FIO benchmark. We launch VM clients on different nodes to perform FIO benchmark and use Sheepdog as the storage

system. To avoid the impact of caching, we use *direct IO* pattern when Sheepdog stars, thus no reads can benefit from the cache on either the client side or storage node side.

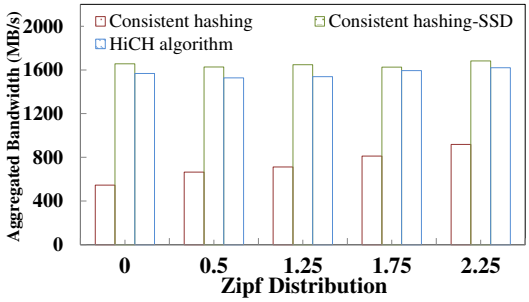
Figure 3 shows the I/O performance of different algorithms. For comparison, we also test the data distribution policy of Sheepdog by only using SSD nodes as storage nodes, namely *consistent hashing-SSD*. We launch 8 clients on different nodes, in which each perform FIO benchmark with 8 jobs and request sizes from 4KB to 1MB. The results were aggregated by adding the performance of each client. From Figure 3(a) and (b), two observations can be made. One is that the performance of three algorithms increases with the increase of request size. Compared to consistent hashing, HiCH and *consistent hashing-SSD* achieve higher throughput from 112% to 238%. As HiCH and *consistent hashing-SSD* use the SSD nodes for object storage, the performance improvement is as expected. The other observation is that the result of HiCH is close to *consistent hashing-SSD*. It confirms the efficiency of our algorithm in a heterogeneous storage system.

The write optimization is often not evident due to the impact of I/O scheduling policy, data consistency, and other factors. However, in our evaluations, it was observed that the HiCH algorithm achieved better sustained bandwidth over the consistent hashing due to the fact that it considers distinct merits of heterogeneous devices. We report the results of one set of representative tests, as shown in Figure 3(c), for the asynchronous sequential write performance with various request sizes. A total 10GB file was written with 64 processes with sync flag. It can be seen that the performance improvement was up to 1.69 times, not as significant as the benefits observed in the read tests, partially due to the fact that the write operation needs to be propagated to all replicated copies.

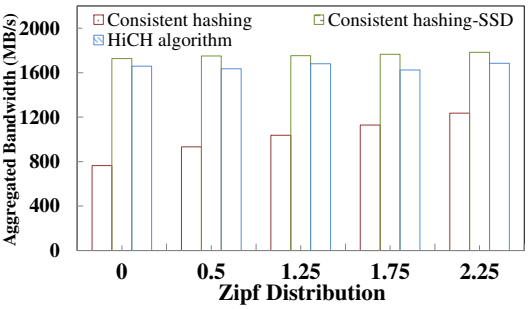
B. Tests with Skewed Data Distribution

The HiCH algorithm uses SSD hashing ring as an additional tier to store frequently read data in a heterogeneous storage system. To validate the performance benefit with HiCH, we use unbalanced data accesses as the workload for evaluation. We run FIO benchmark to measure read operations with increasingly skewed access distribution (Zipf distribution) [24]. The tests were conducted on Sheepdog with 8 VM clients running on different SSD nodes. The performance is aggregated by adding the result of each client, in which the request size of FIO is 256KB. The FIO benchmark generates more skewed data distribution with the increase of Zipf distribution value, meaning that some part of the data are more frequently accessed than others.

Figure 4 shows the read performance by comparing the HiCH algorithm with other algorithms. The *consistent hashing-SSD* means the data distribution policy in Sheepdog only uses SSD nodes as placement. It can be seen that HiCH and *consistent hashing-SSD* achieved higher performance in both random and sequential read operations. This is because that both of them use the SSD nodes for data storage. The HiCH algorithm places the recently accessed data on SSD hashing ring, which can make full use of the throughput of



(a) Ramdom read with increasing skewness



(b) Sequential read with increasing skewness

Fig. 4: Read performance with different data skewness

SSDs. As there is no data replacement in these tests (discussed in Section III(C) and (D)), all the objects are placed on the SSDs. Thus the performance of HiCH in a heterogeneous cluster is close to *consistent hashing-SSD* which uses all SSD nodes for storage. On the other hand, consistent hashing evenly distribute objects on different storage nodes. Although there is slow performance increase with larger skewness, consistent hashing can not leverage the benefits of SSDs well.

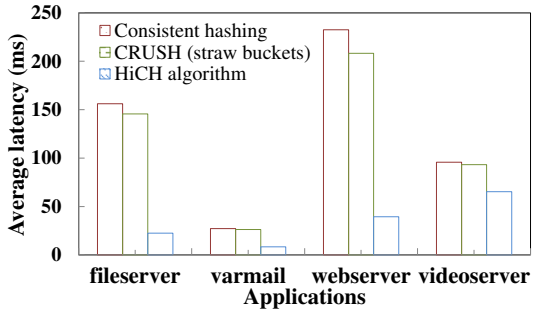
C. File System Workload Evaluation

We evaluate the performance of the HiCH algorithm with the benchmark Filebench [25], which emulates file system level workloads of different real applications. Table I summarizes the profile of emulated file system workloads of four applications. Note that the *WF* means reading or writing a whole file. According to the data amount of workload, we set the watermarks W_h and W_l to 2% and 1% to trigger data replacement. The threshold T_{lag} is set to 10 milliseconds. We launch one client running on a virtual machine on the Sheepdog to collect the experimental results.

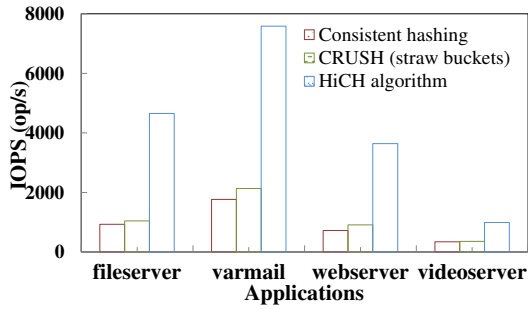
TABLE I: Profiles of emulated file system workloads

Application	Dataset	R/W	Ave File Size	I/O size
fileserver	120GB	1:2	256KB	WF
varmail	65GB	1:1	32KB	WF
webserver	95GB	10:1	256KB	WF
videoserver	136GB	1:0	1GB	1MB

We compare the HiCH algorithm with consistent hashing and straw buckets (a typical replication algorithm in CRUSH [6]) based on Sheepdog. Figure 5 shows the evaluation results of three algorithms under different workloads. It can be seen that the HiCh algorithm achieved best performance



(a) Average latency



(b) IOPS

Fig. 5: Performance under workloads of different applications

for all applications. This is because that HiCH uses SSDs to store objects and make data replacement when necessary. By considering both the frequency and recency, HiCh keeps the hot data on SSD hashing ring. It can improve the entire I/O performance while maintaining load balance according to device capacity. In contrast, the consistent hashing and CRUSH algorithms evenly place data among nodes without distinguishing different device characteristics. The results further confirm the efficacy of the HiCH algorithm in a heterogeneous cluster.

D. System Overhead Analysis

The HiCH algorithm uses two consistent hashing rings to manage HDD and SSD nodes. It maintains the information of data access on SSD nodes with *metaTree* and looks up the object from it when reading or writing an object each time. Besides this, HiCH updates the *metaTree* for data placement. To measure the overhead caused by these operations, we compare the HiCH algorithm with consistent hashing. Figure 6 shows the performance results with running one client on the Sheepdog, which only uses SSD nodes as storage. As expected, the overhead of managing *metaTree* in HiCH is small and can be negligible. As HiCH uses the consistent hashing algorithm for data placement in HDD or SSD hashing ring, it improves the I/O performance with minor additional overhead.

IV. Related Work

There have been numerous research studies in optimizing the heterogeneous storage system with the coexistence of SSDs and HDDs. Welch and Noer [26] propose to optimize a hybrid SSD/HDD HPC storage system based on file size

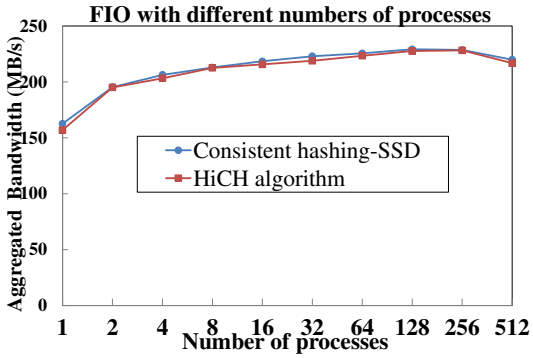


Fig. 6: System overhead for sequential read

distribution. In the proposed scheme, all of the block-level and file-level metadata, and all of the small files are allocated to SSDs, and use HDDs for storage of large file extents. An adaptive multi-level cache (AMC) replacement algorithm is proposed by Cheng et al. [22] to adjust cache level between dynamic random access memory (DRAM) and solid state drive (SSD). They introduce to combine selective Promote and Demote operations to dynamically allocate hot data in DRAM and SSD cache. Zhao and Raicu [27] design a user-lever file system, HyCache, that manages the SSD/HDD heterogeneous storage devices and offers SSD-like performance at a cost similar to a HDD. Distributed systems like the Hadoop HDFS can achieve better performance with HyCache. Feng et al. [28] propose an SSD/HDD hybrid distributed scheme, called HD-Store, for large-scale data, in which the journal file is stored on SSDs in fixed size using append-only mode while segment files focusing on read are stored on HDDs.

The heterogeneous storage architecture has become increasingly popular for massive data storage in data centers. The consistent hashing algorithm can evenly distribute data among nodes, but do not meet the requirements well in a hybrid storage [5]. Some algorithms address data placement in a heterogeneous environment by considering different device characteristics. CRUSH [6] is a scalable pseudo-random data distribution function designed for a tiered storage cluster. The SPOCA [29] and ASURA [30] algorithms can distribute data according to the capacity of each node and can maintain appropriate data placement with minimum data movement when nodes additional or removal. Xie et al. [31] propose a data placement mechanism in accordance to the computing power of each node to improve the task performance in heterogeneous Hadoop clusters. Although these algorithms can effectively distribute data in heterogeneous environments, they focus on one factor while overlooking other characteristics such as throughput of different devices. Other works provide flexible solutions for hybrid stroage cluster by optimizing data placement between heterogeneous nodes [32], [33]. However, these studies lack effective methods to explore heterogeneous node potentials by taking advantage of their distinct merits in a distributed object storage system.

V. Conclusion and Future Work

The consistent hashing algorithm has been widely used in distributed storage systems due to its simplicity and scalability. It uses the hashing functions to map both the data and storage nodes to a hash ring, and places data onto those storage nodes. It provides a key value store and uses hashing functions to assign data to nodes. However, the current consistent hashing algorithm cannot work well in a heterogeneous storage system environment with the co-existence of storage class memory, such as solid state drives, and conventional hard disk drives as it does not distinguish distinct characteristics of heterogeneous storage devices.

In this paper, we propose a hierarchical consistent hashing algorithm (HiCH) specifically for data placement on heterogeneous storage systems. HiCH divides heterogeneous nodes into different buckets and manages a hashing ring for each bucket. It is a hierarchical consistent hashing in which the data in higher level of hashing ring provided an additional storage tier of that of lower level. It places data into various hashing rings according to the hotness, access time, and other information of data access patterns. A data replacement strategy is used to transfer data among hashing rings. We have conducted evaluations based on the Sheepdog, a distributed object-based storage system for virtual machine storage in cloud computing in different scenarios, and evaluations show that HiCH can improve the performance of storage systems, and also make better utilization of heterogeneous storage devices. We plan to further investigate the data distribution and replica management in heterogeneous storage systems to promote better utilization of various storage devices and the productivity of advanced computing systems.

Acknowledgment

This work is supported by the National Science Foundation under grant CNS-1338078 and IIP-1362134 through the Nimboxx membership contribution.

References

[1] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, vol. 41, no. 8, pp. 84–90, 2003.

[2] D. Dai, Y. Chen, D. Kimpe, and R. Ross, "Two-choice randomized dynamic I/O scheduler for object storage systems," in *Proc. of the ACM/IEEE Supercomputing Conference (SC)*, 2014, pp. 635–646.

[3] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," in *Proc. of the 21st ACM Symposium on Operation Systems Principles*, 2007, pp. 205–220.

[4] W. Litwin, M. A. Neimat, and D. A. Schneider, "LH*-a scalable, distributed data structure," *ACM Transactions on Database Systems*, vol. 21, no. 4, pp. 480–525, 1996.

[5] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy, "Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web," in *Proc. of the twenty-ninth annual ACM symposium on Theory of computing*, 1997, pp. 654–663.

[6] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proc. of the 2006 ACM/IEEE Conference on Supercomputing*, 2006, pp. 654–663.

[7] M. Matsumoto and T. Nishimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, Jan. 1998.

[8] W. Xie, J. Zhou, M. Reyes, J. Noble, and Y. Chen, "Two-mode data distribution scheme for heterogeneous storage in data centers," in *Proc. of the 2015 Bigdata conference (Bigdata)*, 2015, pp. 327–332.

[9] J. Zhou, W. Xie, J. Noble, M. Reyes, and Y. Chen, "SUORA: A scalable and uniform data distribution algorithm for heterogeneous storage systems," in *Proc. of the 11th IEEE International Conference on Networking, Architecture, and Storage (NAS)*, 2016.

[10] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan, "Chord: a scalable peer-to-peer lookup protocol for internet applications," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 17–32, 2003.

[11] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.

[12] S. A. Weil, S. A. Brandt, E. L. Miller, and D. D. E. Long, "Ceph: A scalable, high-performance distributed file system," in *Proc. of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006, pp. 307–320.

[13] "Sheepdog Project," 2015. [Online]. Available: http://www.sheepdog-project.org/.

[14] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, "Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches," in *Proc. of the 12th USENIX Conference on FAST*, 2014, pp. 33–45.

[15] J. Darcy, "Hekafs: Multi ring hashing," 2013. [Online]. Available: http://pl.atyp.us/hekafs.org/index.php/2012/07/multi-ring-hashing/

[16] C. Schindelhauer and G. Schomaker, "Weighted distributed hash tables," in *Proc. of the 17th annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2005, pp. 218–227.

[17] Y. Lu, Y. Chen, R. Latham, and Y. Zhuang, "Revealing applications' access pattern in collective I/O for cache management," in *Proc. of the 2014 ACM ICS conference*, 2014, pp. 181–190.

[18] J. Liu, B. Crysler, Y. Lu, and Y. Chen, "Locality-driven high-level I/O aggregation for processing scientific datasets," in *Proc. of the 2013 IEEE International Conference on Big Data*, 2013, pp. 103–111.

[19] "AVL tree," 2015. [Online]. Available: https://en.wikipedia.org/wiki/AVL_tree.

[20] M. T. Ozsu and P. Valduriez, "Principles of distributed database systems," Springer Science and Business Media, Tech. Rep., 2011.

[21] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proc. of the 6th Symposium on Operating Systems Design and Implementation (OSDI)*, 2004, pp. 137–150.

[22] Y. Cheng, W. Chen, Z. Wang, X. Yu, and Y. Xiang, "AMC: an adaptive multi-level cache algorithm in hybrid storage systems," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 16, pp. 4230–4246, 2015.

[23] "Project Voldemort, a distributed database," 2014. [Online]. Available: http://www.project-voldemort.com/voldemort/.

[24] "Zipf Distribution." [Online]. Available: https://en.wikipedia.org/wiki/Zipf's_law.

[25] "File system benchmark." [Online]. Available: http://sourceforge.net/projects/filebench.

[26] B. Welch and G.Noer, "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions," in *Proc. of the 29th Symposium on Mass Storage Systems and Technologies (MSST)*, 2013, pp. 1–12.

[27] D. Zhao and I. Raicu, "HyCache: a user-level caching middleware for distributed file systems," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, 2013, pp. 1997–2006.

[28] Z. Feng, Z. Feng, X. Wang, G. Rao, Y. Wei, and Z. Li, "HDStore: An SSD/HDD hybrid distributed storage scheme for large-scale data," *Web-Age Information Management*, pp. 209–220, 2014.

[29] A. Chawla, B. Reed, K. Juhnke, and G. Syed, "Semantics of caching with SPOCA: A stateless, proportional, optimally-consistent addressing algorithm," in *Proc. of the 2011 USENIX Conference on USENIX annual technical conference*, 2011.

[30] K. I. Ishikawa, "ASURA: Scalable and uniform data distribution algorithm for storage clusters," *arXiv preprint arXiv:1309.7720*, 2013.

[31] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving MapReduce performance through data placement in heterogeneous Hadoop clusters," in *Proc. of the workshop on International Paralel and Distributed Processing Symposium*, 2010, pp. 1–9.

[32] S. He, X. Sun, and A. Haider, "HAS: Heterogeneity-aware selective layout scheme for parallel file systems on hybrid servers," in *Proc. of the 29th International Parallel and Distributed Processing Symposium (IPDPS'15)*, 2015, pp. 613–622.

[33] S. He, X. Sun, Y. Wang, A. Kougkas, and A. Haider, "A heterogeneity-aware region-level data layout for hybrid parallel file systems," in *Proc. of the 44th International Conference on Parallel Processing (ICPP'15)*, 2015, pp. 340–349.