SUORA: A Scalable and Uniform Data Distribution Algorithm for Heterogeneous Storage Systems

Jiang Zhou*, Wei Xie*, Jason Noble†, Kace Echo*, Yong Chen*
*Texas Tech University, Lubbock, TX, USA
†NIMBOXX, Inc., TX, USA
{jiang.zhou, wei.xie}@ttu.edu, {jason.noble}@nimboxx.com, {kace.echo, yong.chen}@ttu.edu

***Abstract*—The data scale in many data centers is growing explosively with emerging applications and usages of big data technologies. Data distribution is a key issue in large-scale distributed storage systems to place petabytes of data or even beyond, among tens or hundreds of thousands of storage devices. In the meantime, heterogeneous storage systems, such as those having devices with hard disk drives (HDDs) and storage class memories (SCMs), have become increasingly popular for massive data storage due to balanced performance, capacity, and cost. Current data distribution algorithms can achieve efficient, scalable, and balanced mapping, but do not distinguish different characteristics of heterogeneous devices well. This paper presents a novel data distribution algorithm called SUORA (Scalable and Uniform storage via Optimally-adaptive and Random number Addressing), to take full advantage of heterogeneous devices. SUORA is a pseudo-random algorithm that uniformly distributes data cross a hybrid and tiered storage cluster. It divides heterogeneous devices, maps them onto different buckets and assigns them to various segments in each bucket. A pseudo-random and deterministic number sequence is generated to map data among segments and devices. Data movement is performed for achieving better read throughput while keeping load balance according to data hotness and bucket threshold. With considering distinct characteristics of heterogeneous storage devices well, the SUORA algorithm achieves a highly efficient adaptive data distribution for data centers and heterogeneous storage systems.**

***Index Terms*—Data distribution algorithm; data placement; data management; heterogeneous storage; data centers**

I. Introduction

With a variety of data center services, Internet applications, and emerging new technologies such as cloud computing and Internet of things, data type and amount are growing with an amazing speed. A large number of semi-structured and unstructured data continue to spring up, leading to the "big data" era. Massive amount of data require effective methods to manage them for meeting demands from different applications. Although various policies are presented for metadata management [1], [2], [3], [4], data placement has become more important than ever as it concerns the capacity, scalability, and performance of the storage system in data centers.

Different from traditional scenarios, most big data storage systems often use a heterogeneous setup to store massive data. Hard disk drives (HDD) are the current dominant storage devices, but are notorious for long access latency and being failure prone. The popular storage class memory (SCM), such as solid state drives (SSDs) and phase change memory (PCM) [5], provide a new promising storage solution with high bandwidth, low latency, and mechanical-component-free, but with inherent limitations of small capacity, short lifetime, and high cost. Table I shows a comparison of characteristics of memory, HDD, and SCM storage devices. Design and development of an innovative hybrid storage to take full advantage of heterogeneous device characteristics is a trend for big data storage solution.

TABLE I: A comparison of characteristics of memory and storage technologies

Media	Access Time (μs)	Endurance[1]	Norm. Cost[2]
DRAM	<0.01	>1E+16	200
PCM	(<0.055) Read, (>0.15) Write	1E+9	24
SSD	(<45) Read, (>200) Write	1E+5	6
HDD	<5000	>1E+16	1

[1] Endurance indicates average write times.
[2] Normalized average cost per GB based on HDD.

The data distribution strategy is mainly to solve the problem of how to place data (files, objects, and blocks) among different devices (racks, nodes, and disks). It often needs to meet some objectives such as fair data distribution, efficient data migration, and load balance. For instance, GPFS [6] divides the file into equal sized blocks and places consecutive blocks on different disks in a round-robin fashion. It improves the I/O throughput with file parallelism but lacks scalability when the node number increases. Consistent hashing functions [7] or pseudo-random algorithms [8] are popularly used in storage systems to efficiently map data or objects to devices, such as in Dynamo [9], Sheepdog [10], and Ceph [11]. They achieve data balance and reduce the amount of data migration when node addition or removal happens. Although these strategies can dynamically distribute data on backend storage, they still lack effective methods to distinguish different devices in a hybrid cluster and place data on them according to their distinct characteristics (capacity, throughput, cost, etc.). Some algorithms address data placement in a heterogeneous environment but they either focus on file stripe layout [12], [13] or data locality [14] to achieve improved performance, or essentially adopt traditional hash functions among devices with homogeneous capacity and weight [15], [16], [17].

Existing data placement algorithms can effective distribute data, but do not meet the requirements well in a *heterogeneous* environment. In addition to conventional objectives, an ideal algorithm for hybrid storage systems should achieve additional

978-1-5090-3315-7/16/$31.00 ©2016 IEEE

goals. *First, it should provide a uniform data distribution by considering distinct characteristics of heterogeneous devices.* It should combine the performance of SCMs with the capacity and economic efficiency of HDDs. *Second, it is desired to achieve optimally-adaptive placement among devices to reduce data migration when device addition or removal happens. Third, data hotness is an important factor to be considered.* Since part of data are frequently read, they should be placed in the devices with better performance. Last but not the least, time and space complexity are important considerations, too. Less calculation time and lower memory footprint will help speed up big data applications on the system with limited resources.

In this paper, we propose a new data distribution algorithm called SUORA (Scalable and Uniform storage via Optimally-adaptive and Random number Addressing), to deal with challenges and to achieve additional goals in an heterogeneous environment as discussed above. SUORA is a pseudo-random algorithm that uniformly distributes data across a hybrid, tiered storage cluster. Different from conventional approaches, SUORA constructs a scalable and unified storage with combination the capacity and performance of different devices. The contribution of this study includes:

- We propose an innovative methodology for distributed data placement among heterogeneous devices to take advantages of their distinct characteristics.
- We design a novel pseudo-random algorithm that effectively and uniformly distributes data cross a hybrid and tiered storage cluster. Combining hotness awareness, the algorithm achieves an adaptive data placement and enhanced read throughput.
- We conduct extensive analysis and evaluation on data distribution and the impact on overall system performance. We also compare the proposed SUORA algorithm with representative distribution algorithms including consistent hashing, CRUSH, and ASURA.

The rest of this paper is organized as follows. Section II discusses related work. Section III describes the SUORA algorithm. Section IV analyzes the SUORA algorithm and presents the evaluation results. Section V summarizes this research and outlines further possible work.

II. Related Work

Numerous active studies have been conducted in recent years on distribution algorithms for data storage.

Stored table management. To establish relations between data and nodes, a stored table is widely adopted in file systems, such as in GFS [2] and HDFS [3]. In table management, combinations of data and storage nodes are memorized in a management table. When accessing the data, the table is searched and the corresponding node is located. Stored table management can easily distribute data among devices, but it requires a large table for lots of data. For example, an HDFS cluster with 6.4 PB data volume with default 64MB block size and 3 replica requires about 72GB memory, which includes the inode information the block belongs to and the datanodes that store the block. Such a large table would require huge amounts of memory. Besides, if only management nodes know that table, every storage node must access the management nodes for data access. Thus, the management nodes become a performance bottleneck.

Hash functions. Hash functions rely on hash methods and specialized algorithms to determine the node corresponding to any data. A simple method to distribute data in a balanced way is round-robin (RR) assignment and the similar improved solution [18]. While these methods are easily to implement, unfortunately there is a large amount of data migration for re-hashing the data when device removal.

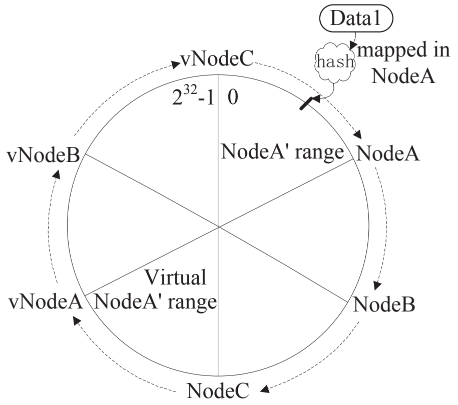
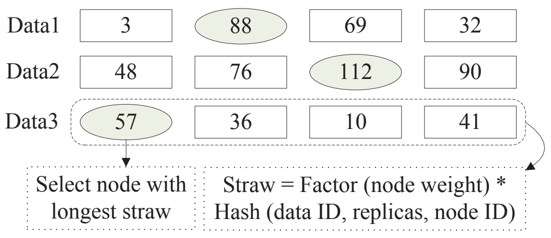


Fig. 1: The mapping of consistent hashing

There have been numerous distributed hash table algorithms and implementations proposed over the years [19], [20], [21]. Consistent hashing [7] algorithm is widely used in distributed systems [9], [10], [22], [23]. It is based on hash functions to construct a hash ring for nodes and map the data on the ring. To achieve better data balance, virtual nodes are applied to distribute data uniformly. The node has more than one hash value and is assigned on the ring with multiple positions and ranges. When a node is added or deleted in the ring, only the data nearby its range will be affected. Figure 1 shows the mapping of consistent hashing, in which each node has a virtual node to balance the placement. Although consistent hashing reaches nearly ideal data load balance and optimal data movement when the node scale changes, it is mainly designed for a homogeneous environment without considering distinct characteristics of different nodes.



	NodeA	NodeB	NodeC	NodeD
Data1	3	88	69	32
Data2	48	76	112	90
Data3	57	36	10	41

Select node with longest straw
Straw = Factor (node weight) * Hash (data ID, replicas, node ID)

Fig. 2: Function selection of Straw Buckets in CRUSH [24]

Another typical algorithm is CRUSH [15], which is based on RUSH [25] family algorithms and used for data placement in Ceph [11]. CRUSH is a scalable pseudo-random, data distribution function designed for distributed object-based storage systems. It divides the cluster into four types of buckets, in which each bucket uses different hash functions. CRUSH provides more flexible data mapping by adopting various buckets and functions in an hierarchical cluster. In the function selection of *straw buckets*, CRUSH draws a straw of random length for each item in the bucket and selects the node with the longest straw to store data. As shown in Figure 2, $Data3$ will be placed on $NodeA$ as $NodeA$ has the largest hash number for the data. It delivers desired data movement between nested items when modified. Although CRUSH provides uniform data placement in a hierarchical cluster, it lacks effective measures to distinguish the device heterogeneity in the buckets.

Hybrid method. Numerous algorithms have been proposed in an attempt to address data placement in an heterogeneous environment [14], [16], [17], [26], [27]. SPOCA [28] is a stateless and optimally-consistent addressing algorithm to place data in a content distribution network (CDN). Each front-end server is assigned a segment of the hash space in a number line proportional to its capacity. The SPOCA routing function uses a hash function to map the request by its name to a point in the line. As not every point in the hash space maps to a front-end server, the result of the hash function may be hashed again till the request lands in an assigned segment. Figure 3 shows a sample assignment of the SPOCA hash map, where a data object is initially hashed to an empty space, but when hashed again, it is assigned to $segment1$. The ASURA algorithm [24] follows the similar idea of SPOCA. It assigns nodes to multiple segments in a number line according to the nodes' capacity. For storing data, the ASURA algorithm generates pseudo-random numbers within a range till one lies within a segment that has been mapped to a server. The ASURA algorithm achieves a trade-off between scalability and efficiency by extending or shrinking the line length for node addition or removal. Although these algorithms can effectively distribute data among heterogeneous devices according to their capacity, they ignore other important characteristics such as throughput of different devices.

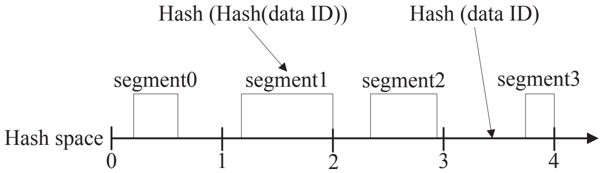


Fig. 3: A sample assignment of the SPOCA hash map

III. The SUORA Algorithm

The SUORA algorithm introduced in this paper distributes data among heterogeneous devices by combining the performance of SCMs with the capacity and economic efficiency of HDDs. It is inspired by the SPOCA [28] and ASURA [24] algorithms, but is more flexible and scalable by taking full advantage of device characteristics. Different from these two existing algorithms, the SUORA algorithm divides heterogeneous devices into buckets and manages them in a tiered architecture. It distributes data among devices according to both their capacity and performance. It essentially generates a *unified* storage management for a heterogeneous environment in that the data is distributed among devices with optimally-adaptive mapping.

A. Algorithm Model

Generally speaking, the SUORA algorithm is designed to reconcile two competing goals: improving the read throughput by making full use of the performance benefits of SCMs and keeping load balance to achieve capacity and economy efficiency of HDDs. To achieve these two goals, SUORA divides heterogeneous devices into different buckets and assigns them to various segments in each bucket. It uses pseudo-random functions to effectively distribute data among segments and buckets. To improve the read performance, SUORA migrates data from HDDs to SCMs in accordance with data hotness. It tends to minimize the computation time and data movement when device addition or removal happens.

We define a multiple dimension-like model in the heterogeneous cluster for SUORA. It divides heterogeneous storage devices into different types of buckets and considers each bucket as a dimension. The bucket represents devices with similar characteristics, such as HDDs or SSDs. Each bucket is associated with a number line that consists of various segments in which one device is assigned to one or more segments. The segment length for a device is calculated according to the device capacity, discussed in Section III(B). Supposing in a heterogeneous cluster, the storage devices are divided into m buckets $\{b_0,\ b_1,\ ...,\ b_{m-1}\}$. For the bucket b_i, there are n segments $\left\{s_{i0},\ s_{i1},\ ...,\ s_{i(n-1)}\right\}$ with their segment length $\left\{l_{i0},\ l_{i1},\ ...,\ l_{i(n-1)}\right\}$. Given a data ID x, SUORA first selects the bucket and then utilizes a series of pseudo-random number generators to map the data on one segment in the bucket. A hash function $f(x,\ e)$ is used to generate random numbers in the range $[u,\ w)$, where e is the seed, and u and w are lower and upper limit of distribution, respectively. It generates a random number sequence $\vec{R} = \{r_0,\ r_1,\ ...,\ r_{n-1}\}$ for the data until it is mapped to one segment. When making replication, SUORA will output multiple buckets and segments for replica placement. The frequently accessed data will be moved between buckets according to bucket threshold values. The threshold value for each bucket $\{v_0,\ v_1,\ ...,\ v_{m-1}\}$ can be predefined or set according to data access patterns. SUORA migrates the hot data to devices with higher throughputs to improve the I/O performance.

Figure 4 shows the model of the proposed SUORA algorithm. As shown in the figure, there are multiple buckets from $HDD_0\ bucket\ b_0$ to $SCM_1\ bucket\ b_6$, which represent devices with higher performance in the clockwise direction. In each bucket, the devices are assigned to segments according to their capacity. The entire storage system can be extended by adding buckets and devices. Initially, all data and replicas are placed in various segments across buckets. If a data

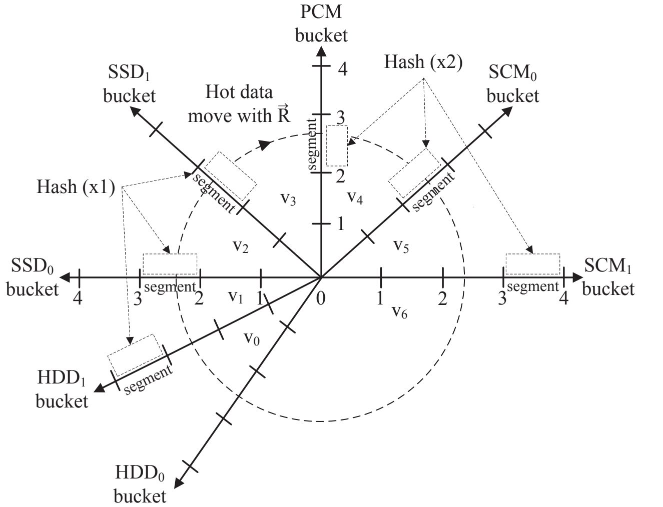


Fig. 4: Model of SUORA algorithm. The threshold v_i is a limit number for hot data movement among buckets.

TABLE II: Node assignment for buckets and segments

Node	Bucket	Capacity	Assigned Segments and Range
A	b_0	1TB	$(s_{00},\ 0,\ 1)$
B	b_0	1.5TB	$(s_{01},\ 1,\ 2),\ (s_{02},\ 2,\ 2.5)$
C	b_0	0.8TB	$(s_{03},\ 3,\ 3.8)$
D	b_1	0.6TB	$(s_{10},\ 0,\ 1)$
E	b_1	0.3TB	$(s_{11},\ 1,\ 1.5)$
F	b_1	0.8TB	$(s_{12},\ 2,\ 3),\ (s_{13},\ 3,\ 3.3)$

item has only one replica, it is placed in the first bucket, $HDD_0\ bucket\ b_0$ in the figure; otherwise, SUORA selects rep buckets for replications with the replica number rep. When making replication, SUORA places the first replica on the bucket with lower performance and larger capacity, such as HDD buckets. The rest replicas will be placed to different buckets clockwise according to their performance, as shown in Figure 4 from SSD buckets to SCM buckets. A random number sequence $\vec{R}$ is generated for each data till it fits one segment in the bucket. It can be seen that data $x1$ and data $x2$ have three replicas that are placed from $HDD_1\ bucket$ to $SSD_1\ bucket$ and from $PCM\ bucket$ to $SCM_1\ bucket$, respectively. When a data item is frequently read and becomes hot, it will move among buckets. Each bucket has its threshold with the values $\{v_0,\ v_1,\ ...,\ v_6\}$. The threshold indicates a limit number for the current bucket, in which the data will move from a previous bucket to it in clockwise direction if the data's hotness number h exceeds the threshold. For example, the data in $HDD_0\ bucket$ will move to $HDD_1\ bucket$ if the data's hotness value h is larger than v_1. The data can be mapped in segments of the new bucket according to the same $\vec{R}$. As the frequently read data are placed in devices with higher performance, such an approach can significantly improve the read throughput for the storage system.

Our SUORA algorithm and model differ from using faster storage devices as multi-level storage cache in two-fold. First, it is not an inclusive setting in which the data in higher level of storage cache hierarchy is a subset of that of lower level. The SUORA algorithm considers different characteristics of heterogeneous devices and achieves uniform data distribution. Second, in a multi-level storage cache design, all writes to a lower level in the hierarchy will go through intermediate cache levels (SSDs or SCMs), which can reduce the lifetime of SSDs or SCMs. With the algorithm model discussed above, we introduce the SUORA algorithm design and implementation in the rest of this section. It can achieve scalable and flexible data placement via the pseudo-random distribution and reduce data migration when device addition or removal happens.

B. Data Distribution Algorithm

The $SUORA$ algorithm divides heterogeneous devices into buckets and places data among the buckets. For convenience and simplicity, we use two buckets to illustrate the design of the algorithm. Suppose the storage devices are a hybrid cluster with nodes equipped with HDDs and SSDs, there are several steps in the algorithm for data distribution.

First, the nodes are divided into two buckets in opposite directions: HDD bucket b_0 and SSD bucket b_1. Each bucket is associated with a number line containing the nodes with similar characteristics.

Second, all nodes in the bucket are assigned to segments in the number line. The segment begins with the point of an integer number with the maximal length set to 1. Each node is assigned to one or more segments considering its capacity through dividing it by a capacity parameter p, which can be predefined. When the segment length of a node exceeds one segment, it is assigned to a new consecutive one with the smallest segment number in the number line. The assignment of segments for storage nodes is performed when the system starts up or nodes are added or removed. At starting up, the segments are assigned through the overall capacity of the node. During data placement, the data will be distributed proportionally in different segments. When node addition, the segment length of the new node will be adjusted according to the remaining capacity of existing nodes. It aims to deal with the situation that new storage nodes are added for extending the total capacity. Table II and Figure 5 show an assumed system and the corresponding mapping of nodes and segments. For example, $(s_{00},\ 0,\ 1)$ means that node A is assigned to segment s_{00} in bucket b_0 with length range $l_{00}(0,\ 1)$. The segment length of each node is computed by the formula with the p being 1TB in b_0 and 0.6TB in b_1.

$$Segment\ length = \frac{Node\ capacity}{p} \tag{1}$$

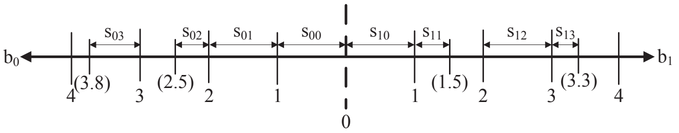


Fig. 5: Mapping of nodes and segments

Third, the data are distributed among nodes with pseudo-random hash functions. Supposing one replica is desired for the data and all items are placed in the HDD bucket b_0 initially. As there may be gaps between nodes in the segment (for the segment length is different or the node may be removed), a random number sequence in a given range is generated according to the data ID till it fits the range of one segment. Algorithm 1 details the initial data distribution in $SUORA$.

Algorithm 1 Initial data distribution in the first bucket b_0

INPUT: data ID x, segment number n, seed e;

```
$segment[n]$ = segment array
$val \Leftarrow hash(x, e)$
while $x$ does not belong to any segment do
    for $i = 0$; $i < n$; $i++$ do
        if $val \in segment[i]\ range$ then
            $segment[i] \Leftarrow x$
            $node\ assigned\ to\ segment[i] \Leftarrow x$
        end if
    end for
    $val \Leftarrow hash(x, e)$
end while
```

When the data are placed in the HDD bucket b_0, they are mapped to nodes according to their hash values. Figure 6 shows the initial distribution, in which four data items with IDs 1 to 4 belong to different segments. With the pseudo-random generators, the random number sequence $\vec{R}$ of each data item is shown as below. The numbers are generated until the value matches one segment in the number line of b_0.

$\vec{R}_{data1} = 4.2,\ 0.9$
$\vec{R}_{data2} = 2.7,\ 1.6$
$\vec{R}_{data3} = 4.8,\ 2.8,\ 2.1$
$\vec{R}_{data4} = 3.9,\ 4.6,\ 3.5$

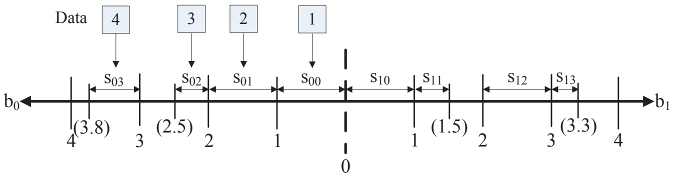
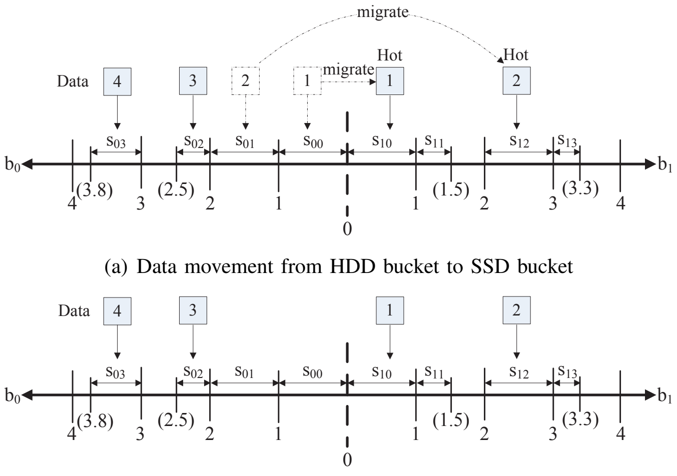


Fig. 6: Initial distribution in HDD bucket

At last, data distribution is automatically adjusted between the HDD and SSD buckets according to the hotness and bucket threshold. The node assigned to each data may change with different access patterns. The frequent read data will be moved from b_0 to b_1 when its hotness exceeds the value v_1. When migrating from the HDD to SSD bucket, a data item is mapped to the segment according to the same random number sequence $\vec{R}$. Figure 7 shows the placement of data before and after their hotness reaches a threshold. From the figure, it can be seen that $data1$ and $data2$ with hotness value exceeding the threshold are moved. With the previous generated $\vec{R}$, the first mapped segment of them in $b1$ is s_{10} (($\vec{r}_1 = 0.9$ for $data1$) and s_{12} ($\vec{r}_0 = 2.7$ for $data2$)), respectively. If no random number matches any segments, new numbers will be generated subsequently until they fit one.

As the data access frequency varies, the hot data tends to move to the bucket with higher performance. On the other hand, the data may be migrated to the bucket with lower performance and larger capacity if there is no storage space in current bucket. By this way, the $SUORA$ algorithm achieves an adaptive mapping by combining the performance of SCM with capacity and economy of HDDs. It ensures load balance and reduces the overhead for recalculation with the same random number sequence $\vec{R}$ when mapping one data.



(a) Data movement from HDD bucket to SSD bucket

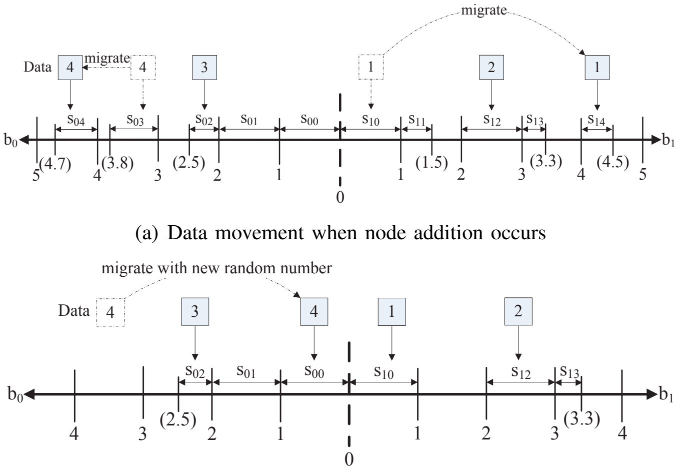
(b) Data placement after adjustment

Fig. 7: Data adjustment for hot data

C. Hotness and Data Movement

Numerous methods or functions can be used for hotness computation [29]. It is not the focus of this study to address which one is the best. To identify the hot data, one method can be used is the Bloom-filter method, i.e. to use a hotness table to compute and store hotness number for the data. The hotness table is an array which keeps the read count of each data item. When a data item is read each time, its ID will be hashed to index in the table and increase the corresponding read counter by one. For reducing the hash collision, multiple hash functions can be used to map data ID in the hotness table.

Besides data movement between the HDD bucket and SSD bucket, the data may be migrated when nodes are added or removed. For each bucket, data movement only occurs inside it when the storage scale is changed. When adding a new node, if there is a random number in the $\vec{R}$ pointing to the new segment prior to the current segment, the data move to the new segment. Otherwise, the data remain in its original position. When removing a node, new random numbers are generated for data on it to be moved to other nodes. Figure 8 shows data movement when node addition and removal occurs after the placement described in Figure 6(b). In Figure 8(a), there are two nodes s_{04} and s_{14} added in b_0 and b_1 with each occupying one segment length $l_{04}(4,\ 4.7)$ and $l_{14}(4,\ 4.5)$, respectively. $Data1$ and $data4$ move because their random numbers $\vec{R}$ ($\vec{r}_0 = 4.2$ for $data1$ and $\vec{r}_1 = 4.6$ for $data4$) fall into the new segments when the new nodes are added. In Figure 8(b), node C (segment s_{03}) and node E (segment s_{11}) are removed from buckets b_0 and b_1. It can be seen that $data4$ placed in the segment s_{03} moves to the segment s_{00} with a



(a) Data movement when node addition occurs

(b) Data movement when node removal occurs

Fig. 8: Data movement when node addition and removal occur

new random number $\vec{r}_3 = 0.8$ as its $\vec{R}$ does not fall into any current segment. Such an approach ensures load balance and reduced data migration.

D. Random Number Functions

SUORA uses the pseudo-random function to generate a random number sequence $\vec{R}$ for each data till it falls into one device. It is based on the data ID x and seed e to generate the $\vec{R}$ in a given range $[u,\ w)$. The pseudo-random number generator has the homogeneity characteristics as described in [24].

With device addition or removal in the bucket, the range of $\vec{R}$ may change to fit segments as the segments cover a wider or narrower area. Simultaneously, the hot data may be migrated from one bucket to another with different segment lengths. Different from ASURA, our algorithm extends or shrinks the number range by multiple pseudo-random number generators among buckets. Each generator uses different seeds to generate $\vec{R}$ in a range. When the number in $\vec{R}$ is larger or less than the given range, it will be substituted by other numbers in the corresponding range. The order of the original random numbers remains unchanged, which ensures a nearly homogeneous distribution in the number line. Suppose the number line of b_1 in Figure 5 is extended from $[0,\ 4)$ to $[0,\ 8)$ and $[0,\ 12)$ for twice. $Data5$ has its initial $\vec{R1}$ and is placed in s_{01} of b_0. When it is moved to b_1 due to hotness, there is no a matching number to map its segment. Then two other random number sequences are generated as below to extend the range for fitting segments in b_1.

$\vec{R1}_{data5} = 3.9,\ 1.6\ \in [0,\ 4)$

$\vec{R2}_{data5} = 7.8,\ 1.4,\ 5.8\ \in [0,\ 8)$

$\vec{R3}_{data5} = 11.6,\ 2.3,\ 10.1,\ 3.6,\ 8.2,\ 4.5\ \in [0,\ 12)$

Combining these three random number sequences, the final $\vec{R}$ in range $[0,\ 12)$ for $Data5$ is as below. Among them, number 7.8 and 5.8 come from $\vec{R2}$ and 3.9 comes from $\vec{R1}$.

$\vec{S}_{data5} = 11.6,\ 7.8,\ 10.1,\ 3.9,\ 8.2,\ 5.8$

By this way, the random number sequence can be extended to different segments and buckets for distribution. When the device is removed and the random number is shrunk, only unnecessary pseudo-random number generators and sequences are eliminated. It ensures the scalability of data placement when the device scale changes.

IV. Evaluation

In this section, we present the evaluation results of the proposed SUORA algorithm by comparing it with typical data distribution algorithms, including consistent hashing [7], straw buckets in CRUSH [15], and ASURA [24]. The evaluation was conducted with trace-based simulations, similar to the evaluation mechanism in the CRUSH and ASURA studies. The trace-based evaluation mechanism is also widely used in many other studies. The evaluation and comparison primarily focus on the new SUORA algorithm and existing consistent hashing, CRUSH, and ASURA algorithms. There is no trace-based simulation and comparison with the SPOCA algorithm because the SPOCA algorithm is primarily used for CDN. Additionally, the ASURA algorithm represents the core idea of the SPOCA algorithm for data distribution in storage systems.

A. Algorithm Analysis

Many existing distribution algorithms can provide scalable mappings and pseudo-equally data distribution. But they lack an effective method to place data in a heterogeneous environment. SUORA is mainly designed to take advantage of heterogeneous devices while maintaining the desired features of distribution algorithms. We evaluate different algorithms based on the analysis from four aspects as described below.

1) **Computation time.** Suppose there are total u nodes and v virtual nodes, the time complexity of the device initialization and data distribution of consistent hashing are $O((u+v) \times \log(u+v))$ and $O(\log(u+v))$, respectively with a balanced binary search tree management (mainly for looking up a node). The straw buckets algorithm in CRUSH does not initialize device configuration and can make a data placement during runtime, in which the time complexity is $O(u)$. For ASURA and SUORA algorithms, their calculation time for device initialization can be ignored as they simply compute the segment length with device capacity. Both of them achieve $O(1)$ for data distribution as the maximum expectation number of times that random numbers need to be generated to fit a segment depends on a constant value [24]. For SUORA, it maintains the hotnesss number additionally. But the time is negligible as the hotness number can be located and counted directly in the hotness table with the index generated by hash functions at runtime.

The calculation time of device initialization is stable as the process is only performed once with the configured storage. When distributing data, the calculation time of consistent hashing and straw buckets algorithms are increased logarithmically and linearly, respectively with the addition of nodes. For consistent hashing, more virtual nodes indicate higher data balance among devices but result in more calculation time. Although optimizations can reduce the expected run time of a hash computation to $O(1)$ [7], they result in a high

TABLE III: Analysis evaluation of different algorithms

Algorithm	Computation time		Memory usage	Uniform distribution		Adaptive placement	
	Device initialization	Data distribution		Homogeneous	Heterogeneous	Device changes	Hot data
Consistent hashing	*Poor* $O((u+v)\times\log(u+v))$	*Fair* $O(\log(u+v))$	*Good* $O(u+v)$	*Poor* Double variability	*Poor* By near capacity	*Excellent* Minimal data move	*Poor* Ignore
Straw buckets in CRUSH	*Excellent* Negligible	*Poor* $O(u)$	*Good* $O(u)$	*Good* Single variability	*Poor* By near capacity	*Excellent* Minimal data move	*Poor* Ignore
ASURA / SPOCA	*Excellent* Negligible	*Good* $O(1)$	*Good* $O(u)$	*Good* Single variability	*Fair* By capacity	*Excellent* Minimal data move	*Poor* Ignore
SUORA	*Excellent* Negligible	*Good* $O(1)$	*Good* $O(u+\epsilon)$	*Good* Single variability	*Excellent* By capacity, throughput, cost, etc.	*Excellent* Minimal data move	*Excellent* Adaptive migration

cost to migrate data when nodes change. Unlike consistent hashing and straw buckets, SUORA and ASURA algorithms keep an invariable time for data distribution whenever the node scale changes. But ASURA lacks an effective method to distribute data amongst heterogeneous devices, which is the key advantage of our proposed algorithm.

2) **Memory consumption.** To distribute data, the algorithm needs to keep relevant information in memory. For consistent hashing, it will maintain node and virtual node ID or hostname and their hash values, in which the space complexity is $O(u+v)$. Straw buckets algorithm only memorizes n node ID and the hash value can be calculated on the fly. In ASURA algorithm, the node ID and its segment length are kept to map data. The random number sequence can be generated when necessary, in which the space requirement is $O(u)$. The SUORA algorithm places data on multiple buckets, in which each bucket maintains different device and segment information. Besides, SUORA maintains read counters in the hotness table. The table memory can be preallocated with limited size as the proportion of hot data is small in storage systems. Thus, the memory requirement of SUORA is $O(u+\epsilon)$, where $O(\epsilon)$ is the memory consumption for the hotness table.

3) **Uniform distribution.** Current algorithms can provide uniform distribution in a *homogeneous* environment, but not in a *heterogeneous* environment. Consistent hashing is a distance-based algorithm, which assigns an address to each node and assigns the data to the node with the address closest to its hash value. It treats each node as the same one and provides a fair data placement. As the hash numbers of nodes and virtual nodes on a ring have variability, as do those of data, the consistent hashing suffers from double variability for distribution. The other three algorithms use hash values of nodes or pseudo-random numbers to map data among devices. They all achieve better uniform distribution with single variability. In a heterogeneous environment, it may result in data unbalances or data skew if distinct features of different devices are not well considered. For example, the HDD with large capacity may have not enough data while the SSD with high throughput has lots of unfrequent access data. Consistent hashing and straw buckets algorithms can approximately consider node capacity by adjusting virtual node numbers or hash values. The ASURA algorithm can adapt the segment length to reflect the node capacity. Although they consider one factor for heterogeneous devices, other device characteristics such as throughput and cost are not measured. Different from them, the SUORA algorithm divides heterogeneous devices into buckets and segments to distinguish their capacity and throughput. It can achieve better uniform data distribution by taking advantages of various device characteristics.

TABLE IV: The specification of devices in the cluster

Device name	Bucket type	Capacity (GB)	Average throughput (MB/s)
Raw WD hard disk	b_0	4000	95
Raw Seagate hard disk	b_1	2000	176
WD Red RAID5 with 4 disks	b_2	1000	263
Samsung 850 EVO	b_3	512	540
Intel P3500	b_4	400	1800

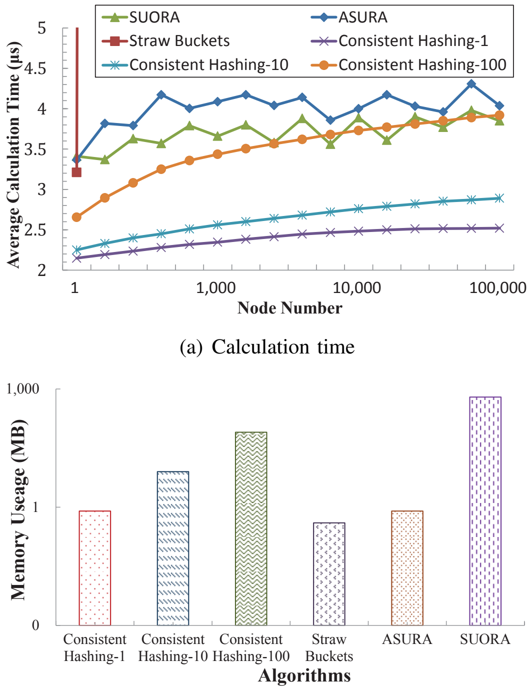
4) **Adaptive placement.** All algorithms can avoid unnecessary data movement when node addition or removal happens. Among them, only the SUORA algorithm considers the hot data and migrates them among buckets to achieve an adaptive placement for heterogeneous devices.

Table III summarizes the analysis evaluation and the comparison of different algorithms.

B. Algorithm Performance

This subsection focuses on understanding the performance of different algorithms. Except consistent hashing, the device initialization time of other algorithms is negligible. Unlike hash functions, the ASURA and our proposed algorithm adopt pseudo-random generator to map data among devices. As the random number can be considered as a hash number from a specified seed, it can also be used for a hash value. For a fair comparison, we choose SIMD-oriented Fast Mersenne Twister (SFMT) [30], a fast pseudo-random number algorithm, to generate both random numbers and hash values. The consistent hashing uses the binary tree search to map data on the node. The simulation test was performed in a cluster with three storage nodes with each having two 8-core E5-2650v2 processors, 128GB memory and one or more of the devices as list in Table IV.

Figure 9 shows the algorithm performance by supposing that the node number varies from 1 to 100,000 with a maximum total of 100PB storage volume in which the data are divided into numerous items (nearly 1.56 billion items) with 64MB each. The vertical axis is scaled with logarithmic values. Different IDs of data items are generated by random number generator. For ASURA and SUORA algorithms, the nodes



(a) Calculation time

(b) Memory consumption

Fig. 9: Performance of different algorithms. The vertical axis is logarithmic scale.

are assigned to segments in a number line sequentially in which the latter uses two buckets with each having half nodes. The range of random numbers is initially set to $[0,\ 16)$ and doubled to extend each time. The *Consistent hashing-v* means each node has v virtual nodes. All the data are placed with one replica. From Figure 9(a), it can be seen that the calculation time of straw buckets increases linearly with the addition of node numbers. This is because it recalculates the hash value for each data item when adding a new node. Compared with consistent hashing, there is a little performance degradation in the ASURA and SUORA algorithms. Random number regeneration for range extension spends more time on the computation. The proposed SUORA algorithm spent less time than ASURA as it places all data on half nodes (in the first bucket) for the initial distribution. It reduces the random number regeneration times and takes advantage of device characteristics, such as half nodes having larger capacity. In Figure 9(b), most algorithms require a low memory footprint less than 100MB supposing both the node ID and hash number have 4 bytes. For the SUORA algorithm, it needs additional memory (<1GB) for hotness table to maintain read counter. The cost is negligible in such a large-scale storage system.

C. Unified Distribution and Throughput

In this subsection, we evaluate data distribution uniformity and throughput in a heterogeneous environment. As the calculation time of straw buckets grows linearly when a new node is added, we mainly compare the other three algorithms which are realistic choices for large-scale storage systems.

TABLE V: Data access pattern

FIO-randread[2]	percentage %	55.58	19.8	7.99	9.93	6.70
	hotness	0	1-25[1]	26-280	281-305	306-659
FIO-randwrite	percentage %	64.83	17.44	8.26	4.71	4.76
	hotness	0	1-2	3-10	11-22	23-313
FIO-randrw	percentage %	65	17.73	8.28	4.76	4.23
	hotness	0	1-18	19-280	281-300	301-792
IOZONE-read	percentage %	64.28	18.29	4.49	7.67	5.27
	hotness	0	1-20	21-50	51-75	76-812
IOZONE-randrw	percentage %	60.81	15.42	7.59	8.46	7.72
	hotness	0	1-15	16-90	91-115	116-784
IOR-read	percentage %	63.94	16.33	7.41	8.25	4.07
	hotness	0	1-20	21-100	101-150	151-819

[1] 1-25 means there is 19.8% data with the read number between 1 and 25, which is also used as setting for bucket threshold values in this pattern.

[2] The bucket threshold values of $\{b_0,\ b_1,\ b_2,\ b_3,\ b_4\}$ can be set to $\{0,\ 1,\ 26,\ 281,\ 306\}$ for FIO-randread trace.

In the tests, we set the bucket threshold values according to real data access pattern. To trace the data access pattern, we deployed the Sheepdog [10], a distributed object storage system, on the cluster and performed benchmarks on a QEMU virtual machine running it. We modified the Sheepdog code and traced I/O requests from the gateway component. Table V shows the data access pattern under different benchmarks. Each benchmark uses a 10GB file as input and sets 4KB for block or record size. The hotness and percentage indicate read times and the proportion of data with related hotness, respectively. Supposing in SUORA, the nodes are divided into five buckets from b_0 to b_4 with different bucket thresholds being set according to different patterns and consists of one type of device as shown in Table IV. For example, data will move from b_0 to b_1 when its hotness exceeds 1 in *FIO-randread* workload pattern. Other algorithms do not distinguish buckets but use the same configuration.

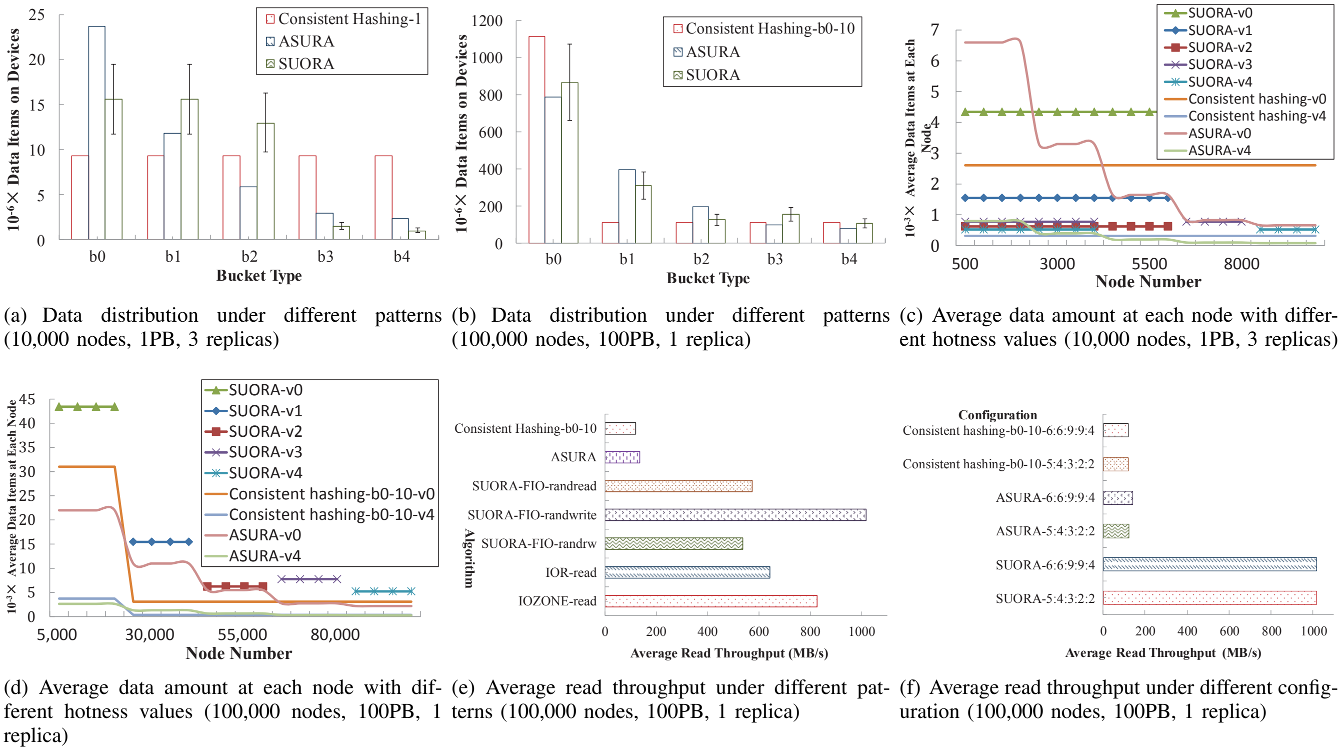
To understand the statistics, we first formulate the equation for data distribution. We assume that each type of bucket has devices with the same average throughput, which is $t_0,\ t_1,\ ...,\ t_{m-1}$. The total data amount of devices in each type of bucket can be represented as $d_0,\ d_1,\ ...,\ d_{m-1}$. For the data d_i in the bucket i, different hotness percentages and hotness numbers are $p_{i0},\ p_{i1},\ ...,\ p_{i(l-1)}$ and $h_{i0},\ h_{i1},\ ...,\ h_{i(l-1)}$, where l is the number of hotness threshold types. The SUORA algorithm places data according to the hotness and bucket threshold, in which the node or segment k in bucket b_i has data amount:

$$D_{SUORA} = d_i \times \frac{l_{ik}}{\sum_{j=0}^{n-1} l_{ij}} \tag{2}$$

Consider the various throughput of devices and read times of data, the average read throughput of the storage is:

$$T_{average} = \frac{\sum_{i=0}^{m-1}\sum_{j=0}^{l-1} d_i \times p_{ij} \times h_{ij}}{\sum_{i=0}^{m-1} \frac{\sum_{j=0}^{l-1} d_i \times p_{ij} \times h_{ij}}{t_i}} \tag{3}$$

Figure 10 shows the data distribution and throughput under different data access patterns as listed in Table V. Except Figure 10(f), each bucket has the same node number. The



(a) Data distribution under different patterns (10,000 nodes, 1PB, 3 replicas)

(b) Data distribution under different patterns (100,000 nodes, 100PB, 1 replica)

(c) Average data amount at each node with different hotness values (10,000 nodes, 1PB, 3 replicas)

(d) Average data amount at each node with different hotness values (100,000 nodes, 100PB, 1 replica)

(e) Average read throughput under different patterns (100,000 nodes, 100PB, 1 replica)

(f) Average read throughput under different configuration (100,000 nodes, 100PB, 1 replica)

Fig. 10: Data distribution and throughput

data are divided to data items with 64MB each. Supposing consistent hashing and ASURA algorithms place multiple replicas according to their original methods like mapping one data copy. Figure 10(a)-(b) show data distribution in each bucket. *Consistent hashing-*b_0*-10* means that consistent hashing is configured with each node having 10 virtual nodes in b_0, in order to store the total data amount. It can be seen that consistent hashing has a nearly fair distribution. It achieves data balance but places excessive data on devices with less capacity. Even using virtual nodes, consistent hashing occupies too much capacity on b_0 but omits other devices. The ASURA algorithm distributes data among devices while considering the capacity proportionally but does not consider the device performance. For the SUORA algorithm, there is few distribution variability when the data access pattern is changed. Compared with consistent hashing and ASURA, the SUORA algorithm not only places most data on buckets with larger capacity but also maps the frequent read data on buckets with higher throughput.

To further understand the algorithm uniformity, we count average data amount on each node under *FIO-randread* pattern, as shown in Figure 10(c)-(d). The data move according to bucket thresholds from $v_0 = 0$ to $v_4 = 306$, e.g., *SUORA-*v_0 means data placement with hotness value between $v_0 = 0$ and $v_1 = 1$ in SUORA. It can be seen that the SUORA algorithm achieves a more efficient adaptive distribution compared with others. It distributes most frequent read data (hotness $> v_4$) in b_4 bucket (node number is from $8,000$ to $10,000$ and from $80,000$ to $100,000$, respectively) to improve the read performance. In Figure 10(c), both b_0, b_1 and b_2 have data with different read frequency. It is because that the SUORA algorithm places replicas on them and only migrates data from b_2 to b_3 or b_4 until the hotness exceeds v_3. It significantly reduces the data movement amount (less than 6%). In contrast, consistent hashing evenly places the data among all devices regardless of hotness. Although there is a little variability in data placement for frequency, the ASURA algorithm lacks an effective method to distinguish different devices.

Figure 10(e)-(f) show the average read throughput under different patterns and configurations. Obviously, the average performance is related with the throughput of each bucket. In Figure 10(e), it can be seen that the SUORA algorithm achieves the best performance. The average read throughput of SUORA is nearly improved from 3.9 to 8.5 times compared to consistent hashing and ASURA algorithms. Specially, the average performance in *FIO-randwrite* pattern is reaching more than half value of the performance of b_4. This is because that the SUORA algorithm uses the devices with the best performance to store data that are read most. The throughput of consistent hashing and ASURA algorithms is uncorrelated with patterns. For them, the performance is mainly affected by virtual node numbers and device capacity, respectively. Figure 10(f) shows the performance when using different configurations under *FIO-randwrite* pattern. For example, *SUORA-6:6:9:9:4* means the ratio of node number in each bucket is *6:6:9:9:4*. Except the ASURA algorithm, the change of node configuration does not affect the overall performance. Evaluation results show that the SUORA algorithm significantly improves the overall performance in different scenarios. Though the percentage of hot data in a practical storage system may be small, it can significantly improve the overall performance with our SUORA algorithm.

V. Conclusion and Future Work

In this research, we introduce the design of a new data distribution algorithm called SUORA for heterogeneous storage systems. The SUORA algorithm addresses the challenges in balanced performance, capacity, and cost for data placement by taking advantages of heterogeneous device characteristics. It divides heterogeneous devices into multiple buckets and assigns them to different segments in each bucket. The bucket can reflect the underlying distinct device characteristics and benefit for data movement based on data hotness and bucket threshold. The SUORA algorithm uses a pseudo-random number sequence to fairly and uniformly distribute the data among devices in the bucket. By combining the performance with the capacity and economic efficiency of different devices, the SUORA algorithm constructs a unified and adaptive storage solution for a heterogeneous environment. In the future, we plan to further explore improvements in data placement. Our further study also includes an effective data I/O pattern detection and data caching strategy combined with the SUORA algorithm in a heterogeneous storage system.

Acknowledgment

This work is supported by the National Science Foundation and the Department of Defense in partnership with the NSF REU program under grant CNS-1263183, by the National Science Foundation under grant CNS-1338078 and IIP-1362134 through the Nimboxx membership contribution.

References

[1] P. H. Carns, W. B. L. III, R. Ross, and R. Thakur, "PVFS: A parallel file system for Linux clusters," in *Proc. of the 4th Annual Linux Showcase and Conference*, Atlanta, USA, Oct. 2000, pp. 391–430.

[2] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," in *Proc. of SOSP'03*, New York, USA, Oct. 2003, pp. 29–43.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. of MSST'10*, Incline Village, USA, May 2010, pp. 1–10.

[4] D. Dai, R. Ross, P. Carns, D. Kimpe, and Y. Chen., "Using property graphs for rich metadata management in hpc systems," in *the 9th Parallel Data Storage Workshop held in conjunction with SC14 (PDSW'14)*, 2014, pp. 7–12.

[5] 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.

[6] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. of the First USENIX Conference on File and Stroage Technologies*, 2002, pp. 231–244.

[7] 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.

[8] M. Matsumoto and T. Nichimura, "Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Transactions on Modeling and Computer Simulation*, vol. 8, no. 1, pp. 3–30, 1998.

[9] 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.

[10] "Sheepdog project," 2015. [Online]. Available: https://github.com/sheepdog/sheepdog/wiki.

[11] 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.

[12] S. He, X.-H. Sun, Y. Wang, A. Koughkas, 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*, 2015.

[13] S. He, X.-H. 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*, 2015, pp. 613–622.

[14] 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.

[15] 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.

[16] A. Brinkmann, K. Salzwedel, and C. Scheideler, "Efficient, distributed data placement strategies for storage area networks," in *Proc. of the 12th Annual ACM Symposium on Parallel Algorithms and Architectures*, 2000, pp. 119–128.

[17] R. J. Honicky and E. L. Miller, "A fast algorithm for online placement and reorganization of replicated data," in *Proc. of the 17th International Parallel and Distributed Processing Symposium*, 2003.

[18] D. M. Choy, R. Fagin, and L. Stockmeyer, "Efficiently extendible mappings for balanced data distribution," *Algorithmica*, vol. 16, no. 2, pp. 215–232, 1996.

[19] J. M. Wozniak, B. Jacobs, R. Latham, S. Lang, S. W. Son, and R. Ross, "C-MPI: A DHT implementation for grid and HPC environments," in *Preprint ANL/MCS-P1746-0410*, 2010.

[20] C. Docan, M. Parashar, and S. Klasky, "Dataspaces: an interaction and coordination framework for coupled simulation workflows," *Cluster Computing*, vol. 15, no. 2, pp. 163–181, 2011.

[21] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, and I. Raicu, "ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table," in *Proc. of the 27th International Symposium on Parallel and Distributed Processing*, 2013, pp. 775–787.

[22] 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.

[23] H. Tang, A. Gulbeden, J. Zhou, W. Strathearn, T. Yang, and L. Chu, "A self-organizing storage cluster for parallel data-intensive applications," in *Proc. of the 2004 ACM/IEEE Conference on Supercomputing*, 2004, pp. 52–63.

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

[25] R. J. Honicky and E. L. Miller, "Replication under scalable hashing: A family of algorithms for scalable decentralized data distribution," in *Proc. of the 18th International Parallel and Distributed Processing Symposium*, 2004.

[26] J. Zhou, W. Xie, Q. Gu, and Y. Chen, "Hierarchical consistent hashing for heterogeneous object-based storage," in *Proc. of the 14th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2016.

[27] 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.

[28] 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.

[29] D. Park and D. Du, "Hot data identification for flash-based storage systems using multiple bloom filters," in *IEEE 27th Symposium on Mass Storage Systems and Technologies*, 2011, pp. 1–11.

[30] "Simd-oriented fast mersenne twister," 2013. [Online]. Available: http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/SFMT/index.html