Data Distribution for Heterogeneous Storage Systems

Jiang Zhou, Yong Chen, Mai Zheng, and Weiping Wang

Abstract—The exponential growth of data in many science and engineering domains poses significant challenges to storage systems. Data distribution is a critical component in large-scale distributed storage systems and plays a vital role in placing petabytes of data and beyond, among tens to hundreds of thousands of storage devices. 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 their distinct and complement characteristics. This paper presents a new data distribution algorithm called SUORA (Scalable and Uniform storage via Optimally-adaptive and Random number Addressing) specifically for heterogeneous devices to maximize the benefits of them. SUORA provides a fully symmetric, highly efficient methodology to distribute data across a hybrid and tiered storage cluster. It divides heterogeneous devices into different buckets and segments, and adopts pseudo-random functions to map data onto them with the balanced consideration of capacity, performance and life-time. By analyzing hotness and access patterns, SUORA gradually moves hot data from HDDs to SCMs to optimize the throughput, and moves cold data reversely for load balance. It combines data replication with migration to significantly reduce movement overhead while making data placement more adaptive to different workloads. Extensive evaluations on simulation and Sheepdog storage system show that, with considering distinct characteristics of various devices thoroughly, SUORA improves the overall performance efficiency of heterogeneous storage systems.

Index Terms—Parallel/distributed file systems, data distribution, data placement, heterogeneous storage, data replication

1 Introduction

The exponential growth of data volume in many science and engineering domains poses constant challenges to storage systems. Many high-performance computing systems and cloud data centers have built infrastructures to host hundreds petabytes of data to accommodate growing needs of their applications. One of the critical challenges large-scale data centers face today is the management of data on a large number of storage nodes. Traditional parallel/distributed file systems, like GPFS [1], Ceph [2], and GFS [3], are widely used to achieve high-performance I/O. Data are striped over storage nodes so that read and write operations can take advantage of high concurrency for better bandwidth. Storage systems usually have dedicated metadata servers to decouple the metadata service (e.g., the namespace service) from data store for better scalability [3, 2]. However, massive data management remains a critical challenge that often limits the I/O performance and storage scalability.

On the other hand, large-scale storage systems often use a heterogeneous setup. Hard disk drives (HDDs) are still the dominant storage devices, but are notorious for long seek time and rotational latency. The storage class memory (SCM) devices gone through tremendous advances in recent years with the development of non-volatile memory (NVM) technologies. For instance, the high-bandwidth, low-latency, and mechanical-component-free characteristics of flash-based Solid State Drives (SSDs) make them rapidly adopted in many storage systems. Further, other NVM technologies, such as phase change memory (PCM) [4], spin-transfer torque RAM (STTRAM) [5], and the recent 3D XPoint [6], are considered a competitive additional tier of storage hierarchy in the near future. All these types of storage devices will form a *heterogeneous* storage hierarchy. Table 1 shows a comparison of characteristics of NVM, SSD, and HDD storage devices with reference to the common product specifications. Although devices exhibit different performance based on the usage of the system, we use the average access time for a consistent comparison among them. Arguably, it is strongly desired to design and develop an efficient *heterogeneous storage solution to take advantages of a variety of devices*, which has a significant impact on future storage systems too.

TABLE 1: Characteristics of heterogeneous storage devices

Device	Avg. latency	Capacity[1]	Endurance[2]	Cost
NVM (STT-RAM)	5∼10ns	<32GB/s	10^{12}	Highest
NVM (PCM)	50ns∼15μs	<256GB/s	10^{12}	$2-8/GB
SSD	35∼350μs	<1TB/s	$10^5 - 10^6$	$0.5-2/GB
HDD	3.5∼5ms	>1TB/s	$> 10^{16}$	$0.06-0.3/GB

[1] Capacity indicates the common maximum capacity of a single memory stick/SSD/HDD disk.
[2] Endurance indicates average write times for the life-time of the device.

The key component in data management is the distribution of data among devices (or nodes). The data distribution (or data placement) strategy establishes the mapping between datasets and devices [7]. It needs to meet objectives such as efficient decision, small amount of data movement, and load balance. For instance, GPFS [1] and numerous other parallel/distributed file systems divide a file into equal-size blocks and place consecutive blocks on different disks in a round-robin fashion. Consistent hashing [8] or pseudo-random algorithms [9] are also popular for mapping data or objects onto devices efficiently, and are widely used in systems like Sheepdog [10] and GlusterFS [11]. They achieve data balance well and only a small amount of data migration occurs when node addition or removal happens. Although these

• *Jiang Zhou and Weiping Wang are with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China. Yong Chen is with the Department of Computer Science, Texas Tech University, Lubbock, Texas, USA. Mai Zheng is with the Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa, USA. E-mail: {zhoujiang, wangweiping}@iie.ac.cn, yong.chen@ttu.edu, mai@iastate.edu.*

strategies can distribute data into storage devices evenly, they are not designed to differentiate characteristics of distinct devices in a heterogeneous environment and place data accordingly and efficiently. Some recent studies attempted to approach this issue but they either focus on file stripe management [12] or performance improvement by exploring fast devices (e.g., SSDs) [13], or essentially adopt traditional hash functions among devices with homogeneous capacity and weight [9].

Data placement in a heterogeneous environment is quite different from a homogeneous environment. Our research studies in this field suggest that a desired data distribution algorithm for heterogeneous storage systems should achieve additional goals. First, it should provide a uniform and adaptive data distribution by considering distinct characteristics of heterogeneous devices. It should combine the performance and endurance of SCMs with the capacity and economic efficiency of HDDs. Second, it is desired to achieve optimally-adaptive placement among devices while reducing data movement cost. Third, it is desired to consider changed access patterns of different applications. For instance, a study [14] shows that hot-data identification for flash-memory storage systems (around 20% as "hot data") strongly affects the performance of flash-memory access and its life-time. Last but not the least, time and space complexity are important considerations too. Less compute time and lower memory footprint will help speed up applications on the system.

In this paper, we introduce a new data distribution algorithm called SUORA (Scalable and Uniform storage via Optimally-adaptive and Random number Addressing) to address challenges and to achieve goals in heterogeneous environments as discussed above. SUORA is a pseudo-random algorithm that uniformly distributes data across a hybrid and tiered storage cluster. It manages heterogeneous devices with buckets and segments, and uses pseudo-random functions to distribute data on them with a balance among capacity utilization, performance efficiency, and wear leveling. Data replication and movement are performed for optimally-adaptive placement according to data-access hotness and pattern. SUORA is designed to efficiently reorganize data and reduce data migration when the access pattern or device member changes. It achieves load balance and has minor memory footprint too. Compared with traditional algorithms such as consistent hashing [8] and ASURA [15], the average read throughput is improved by more than 1.5 and 2 times, respectively, in our evaluation tests. The contribution of this study includes:

- We present a new pseudo-random method to efficiently place data cross a hybrid and tiered storage cluster in a fully symmetric, uniform manner.
- We design a novel data distribution algorithm that unifies the management of heterogeneous devices, and uses pseudo-random functions to distribute data among them by taking full consideration of capacity, performance and life-time.
- We introduce hotness awareness to achieve an adaptive data placement, which gradually moves hot data from HDDs to SCMs according to hotness and access patterns to improve read throughput, and moves cold data reversely for load balance.
- We combine data replication with migration to significantly reduce movement cost and read congestion on SCMs, while also achieving efficient write performance under different patterns.
- We conduct extensive tests based on simulation and Sheepdog storage system to analyze and study the impact on overall system performance. We further compare the proposed SUORA algorithm with representative distribution algorithms including consistent hashing, CRUSH and ASURA, to show the efficiency.

The rest of this paper is organized as follows. Section 2 discusses the background of this study and related work. Section 3 introduces the SUORA algorithm. Section 4 analyzes the SUORA algorithm and presents the evaluation results. Section 5 summarizes this research and outlines further possible work.

2 Background and Related Work

Numerous studies have been conducted in recent years on distribution algorithms for storage systems. We discuss existing work and analyze their merits and shortcomings in this section.

Table-based management method. To establish the position relation between data and storage nodes, a mapping table is widely adopted in file systems, such as GFS [3] and HDFS [16]. In this method, the mapping between data and storage nodes is memorized in a management table. When accessing data, the table is searched and the corresponding node is located. Mapping table management can easily distribute data among nodes, but its memory footprint will significantly increase with the exponentially growth of data. Besides, if only management nodes keep that table, every storage node must communicate with the management nodes for data access. Thus, the management nodes can potentially become the performance bottleneck.

Hash-based management method. In contrast to table-based management, the hash-based management methods do not need to reserve and manage such a large mapping table. These methods rely on specialized hash algorithms to determine the node corresponding to any data.

Consistent hashing [8] is a data distribution algorithm that has been widely used in parallel/distributed file systems [17, 10, 11]. It is based on hash functions to construct a hash ring, a hypothetical data structure that contains a list of hash values, for data and nodes mapping on the ring. As consistent hashing distributes data randomly, virtual nodes are generated to place data more uniformly. Each physical node may have multiple virtual nodes, which are responsible for multiple positions assigned along the hash ring. If a virtual node is selected, the physical node associated with that virtual node is used for data placement. The node capacity can also be considered by adjusting virtual node numbers or hash values [10]. When a node is added or deleted on the ring, only the data nearby its range will be affected. Although consistent hashing achieves impressive data load balance and data movement when the node scale changes, it is primarily designed for a homogeneous environment.

CRUSH is a scalable and pseudo-random data distribution function designed for Ceph system [9, 2]. It divides the cluster into buckets, which can contain any number of devices or other buckets in a storage hierarchy. CRUSH provides four types of buckets, and adopts different hash functions for flexible data mapping. Figure 1 describes the paradigm of straw buckets in CRUSH. With straw buckets, each node has an individual hash value for a data item, and data are stored in the node having the largest hash value for the data. It achieves a small amount of data movement when nodes are added or removed. Although CRUSH provides uniform data placement in a hierarchical cluster, it lacks an effective measure to distinguish the device heterogeneity in buckets.

Other typical algorithms include SPOCA [18], ASURA [15], etc [19]. ASURA is a data distribution algorithm that relies on pseudo random numbers [20] in which the general idea is first

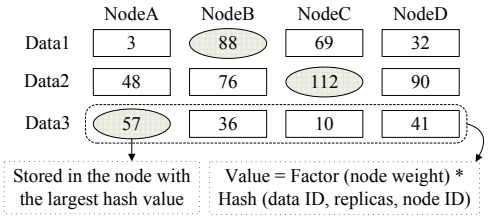


Fig. 1: Function selection of straw buckets in CRUSH

introduced in [18]. ASURA assigns a segment or a series of segments to represent a storage node. A segment is basically a range in a one-directional number line that starts with an integer. The total length of all the segments that are associated with a node represents the capacity of it. For storing data, ASURA generates pseudo-random numbers within a range till one lies within a segment that has been mapped to a server. Figure 2 shows a sample assignment of the ASURA hash map, where a data object is initially hashed to an empty space, and when hashed again, it is assigned to $segment1$. These algorithms can distribute data on heterogeneous nodes, but they mainly focus on one factor (such as capacity) and lack an efficient method to consider multiple characteristics of different devices in a holistic way.

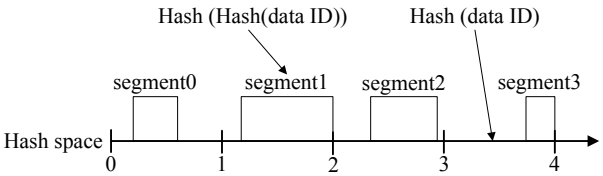


Fig. 2: A sample assignment of the ASURA/SPOCA hash map

Hybrid method. Numerous algorithms have also been proposed as an attempt to address data placement in a heterogeneous environment. Welch et. al. [21] propose to allocate metadata and small files onto SSDs whereas using the much cheaper HDD storage for large files. HiCH [22] manages data with hierarchical consistent hashing rings, in which the SSD ring is used as cache to hold the first copy of data and the HDD ring contains the rest replicas. It uses consistent hashing to evenly distribute data on HDDs or SSDs, and may cause frequent data replacement (i.e., cold data eviction from cache) due to the capacity limitation of SSD ring. OctopusFS [23] stores file blocks on tiered storage, and strikes a trade-off with the consideration of different storage media features via multi-objective optimization. It maintains a mapping table on the metadata server, and moves data if they are over-replicated on some particular tier. HARL [24] presents heterogeneity-aware data layout scheme for parallel file systems. It determines the optimized stripe sizes on HDD servers and SSD servers based on data access cost model. With data replication, HARL can redirect file requests to proper replicas with the lowest access costs. H2DP [25] further devises a dynamic data migration strategy, which moves cold data from hybrid servers to HDD servers and hot data reversely. PRS [26] focuses on replication optimization in heterogeneous environments. It selectively replicates data based on I/O correlation of data accesses, where the first and second replicas of data are placed with default data placement (i.e., via consistent hashing), and the rest are created based on identified patterns. It then adopts the pseudo-random algorithm to proportionally distribute replicas among nodes according to performance.

Different from them, the proposed SUORA algorithm provides a unified and efficient solution for data distribution in hybrid, tiered storage. SUORA uses a *pseudo-random based bucket algorithm* for data placement. It first divides heterogeneous devices into buckets by considering device performance, then it assigns devices in each bucket to segments based on their capacity. With the pseudo-random mapping, SUORA distributes data among buckets and devices with a comprehensive consideration of capacity, performance and life-time. With replication, SUORA initially places data on slow buckets and then conducts data migration according to access pattern analysis. Through comparing hotness and learned bucket thresholds, it gradually moves replicas of hot data from slow buckets to appropriate fast buckets, and distributes them on SCMs according to capacity and life-time. By combining data replication with migration, SUORA can take full advantages of SCMs for performance efficiency while significantly reducing read congestion and movement cost. Cold data migration will also occur for load balance. The evaluation results show that SUORA can make better use of heterogeneous devices and adapt to different workloads. A preliminary study of this research has appeared in [27], and this paper significantly extends the previous research in data distribution, replication and migration and presents a complete study of this subject.

3 The SUORA Algorithm

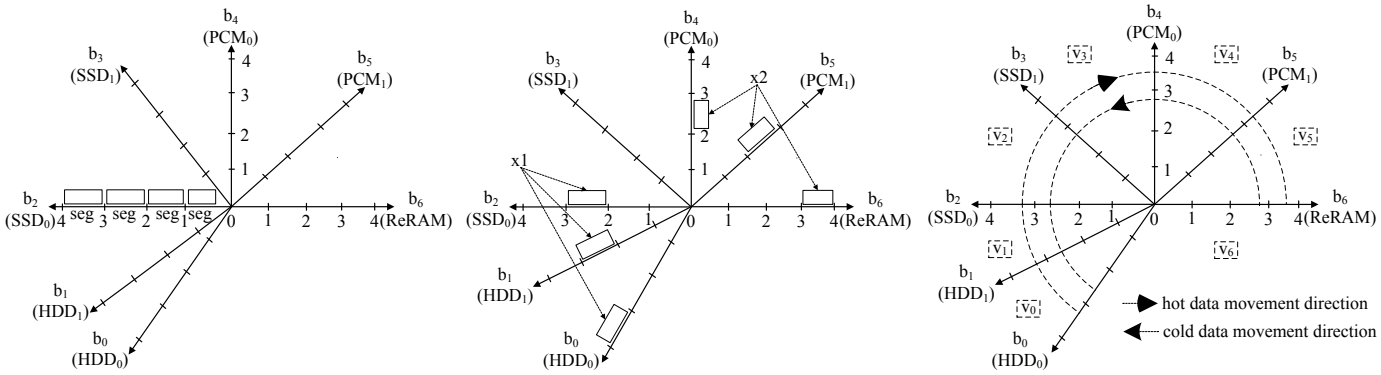
Our goal is to build a unified storage management for a heterogeneous environment where data is distributed among a variety of devices (or nodes). In this section, we will introduce the design and implementation of the SUORA algorithm.

3.1 Algorithm Model

3.1.1 Heterogeneous Devices Management

We define a multi-dimension model for SUORA to uniformly manage devices and distribute data in a heterogeneous environment. This model divides heterogeneous storage devices into different types of buckets and considers each bucket as a dimension. One bucket represents a performance tier with homogeneous devices (with the same throughput/latency), such as HDDs or SSDs. For each bucket, it is further divided into various segments in which a device is assigned to one or more segments according to the device's capacity. Each segment has a range (the range length is calculated in equation (1) as discussed in Section 3.2) and the range of all segments forms a number line, where each range is ordered with the start value, 0. Assuming in a heterogeneous storage system, the storage devices are divided into m buckets as $\{b_0,\ b_1,\ ...,\ b_{m-1}\}$. For the bucket b_i, it has a number line corresponding with it, where there are n segments $\{s_{i0},\ s_{i1},\ ...,\ s_{i(n-1)}\}$, with their segment length as $\{l_{i0},\ l_{i1},\ ...,\ l_{i(n-1)}\}$.

Figure 3 shows the model of the proposed SUORA algorithm. As shown in Figure 3(a), there are multiple buckets: from b_0 (HDD_0 bucket) to b_6 ($ReRAM$ bucket), which represent different types/classes of devices. Note that b_0 and b_1 are both HDD buckets (the similar setting for SSD buckets and PCM buckets) because it is also possible for the same type of devices to have different performance specification. For example, SSDs with PCIe and SATA interfaces can have very different bandwidth and latency, thus they fall into different buckets in our model. The device performance of each bucket increases in the clockwise direction. The bucket with the lowest performance is the "bottom bucket", i.e. HDD_0 $bucket$ in the figure. In each bucket, the devices are assigned to segments (denoted as "seg") according to their capacity. For example, bucket b_2 has four segments (as shown by rectangle) with different range where the maximum range length for each segment is 1. The storage system can scale up or scale down by adding or removing devices and buckets.



(a) Bucket and segment assignment for heterogeneous devices

(b) Data mapping and replication on heterogeneous devices, e.g., data x_1 and x_2 have three replicas

(c) Data movement among different buckets

Fig. 3: Model of SUORA algorithm. In (a) and (b), SUORA divides heterogeneous devices into different types of buckets for *performance feature (throughput/latency)*, with one directed axial coordinate (a number line) representing one bucket. The bucket represents a class of devices with same performance, such as HDD_0, or HHD_1, or SSD_0 class of devices. For each bucket, it is further divided into various segments in which a specific device is assigned to one or more segments for *capacity feature* (e.g., four segments in the SSD_0 class of devices with their range being $[0,\ 1)$, $[1,\ 2)$, $[2,\ 3)$, and $[3,\ 4)$). Each segment has different lengths with the maximum length set to 1, representing the device capacity size. Data are distributed on buckets and segments via pseudo-random hash functions. In (c), hot data will move from slow bucket to fast bucket in the clockwise direction, and cold data (infrequently accessed) will move in the counter-clockwise direction. Each bucket is associated with a threshold, such as v_0 for b_0 and v_1 for b_1, and the threshold is used to decide data movement according to workloads.

3.1.2 Data Mapping in Heterogeneous Devices

Based on the bucket and segment management for heterogeneous devices, SUORA distributes data in two steps. First, given a data with ID x, SUORA selects buckets for the data. If a data item has only one copy, it is initially placed in the slow bucket for capacity utilization goal, such as b_0 in Figure 3(a). If a data item has rep copies, SUORA selects rep buckets from the bottom bucket walking clockwise in the model and makes replicas with a replication algorithm in Section 3.6. The bucket selection will not compromise the performance goal because the data will migrate to fast bucket if it is frequently accessed. It can also reduce data search overhead after data movement as described later.

Second, SUORA utilizes a series of pseudo-random number generators to map the data on one segment in 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 bound 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 with a same sequence $\vec{R}$. Figure 3(b) shows the data mapping and replication in heterogeneous devices. It can be seen that data $x1$ and data $x2$ have three replicas that are placed from HDD_0 bucket to SSD_0 bucket and from PCM_0 bucket to $ReRAM$ bucket, respectively. Note that data $x2$ has no replicas on the slow buckets (e.g. HDD_0 and HDD_1 buckets) with the rationale discussed in detail below.

3.1.3 Data Movement Between Devices

With the bucket and segment assignment, SUORA manages heterogeneous devices in a unified way and places data on them. To make full use of the performance of fast devices, it is critical to consider data-access patterns in different workloads. For instance, "cold" datasets that are not frequently accessed should be stored in slow HDDs while "hot" datasets are placed in fast SCMs. Thus, SUORA moves data between buckets according to their hotness. This means when a data item is frequently accessed and becomes hot, it will be moved from a slow bucket to a fast one. However, moving all hot data to fast devices will cause other problems: congestion in I/O accesses and wear leveling on these flash-based devices. To address this issue, SUORA defines a *threshold* for each bucket to limit the amount of data movement, and considers device life-time when moving data to fast buckets. The data will migrate between buckets by comparing data-access frequency with bucket thresholds, which makes data with different hotness be adaptively distributed on different devices according to real workloads.

Given m buckets, each bucket has its threshold with the values as $\{v_0,\ v_1,\ ...,\ v_{m-1}\}$, respectively. The threshold indicates a limit for the current bucket, in which the data will move from a previous bucket to it in the clockwise direction if the data hotness h exceeds the threshold. The threshold of each bucket can be set according to device specification and data-access patterns (discussed in Section 4.3). An appropriate threshold will reduce the overhead for data movement among different buckets.

Figure 3(c) shows an example of data movement across different buckets. The hot data will move clockwise from HDD_0 bucket to $ReRAM$ bucket, in which the bucket thresholds are $\{v_0,\ v_1,\ ...,\ v_6\}$, where v_0 is set to 0. For example, the data in HDD_0 bucket will move to HDD_1 bucket if the hotness h is larger than v_1. The data can be mapped into segments of the new bucket according to the same random number sequence $\vec{R}$. As these frequently accessed data are placed in devices with higher performance, such an approach can improve the read performance for the storage system.

The SUORA algorithm and its model differ from using fast storage devices as multi-level cache store. In most multi-level storage cache designs, all writes (even not replicas) to a lower level in the hierarchy will go through intermediate cache levels (SSDs or SCMs), which can reduce the life-time of them [28]. Some exclusive cache store provide a hierarchy in which the contents of different levels are exclusive. However, the lower level cache contains only victim or copy-back cache data that are ejected from the higher level due to conflict misses [29]. In contrast, SUORA distributes data among heterogeneous devices by considering their

TABLE 2: 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	$(s_{12},\ 2,\ 3),\ (s_{13},\ 3,\ 3.3)$

distinct characteristics. It initially places data in slow buckets, and gradually moves replicas of hot data to fast buckets based on access patterns for performance benefits.

3.2 Algorithm Design

The SUORA algorithm divides heterogeneous devices into buckets and places data among them. For convenience and simplicity, we use two buckets to illustrate the design of the algorithm. Suppose the storage system is equipped with one type of HDD and one type of SSD, the algorithm takes steps for data distribution.

First, these storage devices are divided into two buckets: HDD bucket b_0 and SSD bucket b_1. Each bucket is associated with a number line containing all devices in one particular type, with the same bandwidth/latency characteristics (the capacity can be different though).

Second, all devices (or nodes) in the bucket are assigned to segments in the number line. To simplify the mapping, the segment begins with the point of an integer number with the maximal length set to 1 (an example is shown in Figure 4). Each node is assigned to one or more segments considering its capacity through dividing it by a capacity parameter p that can be predefined, e.g. the average capacity of nodes in bucket or a specified value. If 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.

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

The assignment of segments for storage nodes is performed when the system starts up. Upon starting up, the segments are assigned through the total capacity of the node. During data placement, the data will be distributed proportionally in different segments. For each time a node is added, we adjust the parameter p to calculate its segment length, such as using the average value of remaining capacity of nodes in bucket. Compared to previous nodes, the new node will have larger segment length if they are the same capacity size. The segments of new nodes will be added along the number line of bucket, where the previous nodes still keep their original segment ranges. As such, the added node will contain more new data, which makes data distribution more proportional to the capacity. When there is no enough space in one bucket, cold data movement will occur for the load balance purpose (see Section 3.4). Table 2 shows an example and the corresponding mapping of nodes and segments. For instance, $(s_{00},\ 0,\ 1)$ means that node A is assigned to segment s_{00} in bucket b_0 with length range $l_{00}(0,\ 1)$. More specifically, 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.

Third, data are distributed among nodes with pseudo-random hash functions. Assuming there are no replicas, all items are initially placed in the HDD bucket b_0 for capacity utilization (the replication algorithm is discussed in Section 3.6). This is not subject to performance degradation because the data will move from HDDs to SSDs in future. As there may be gaps between

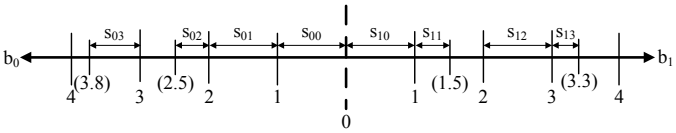


Fig. 4: Mapping of nodes and segments. Two arrowed axes in the opposite directions indicate two buckets b_0 and b_1. Each node is assigned to one or more segments in the bucket.

nodes in the segment (as 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.

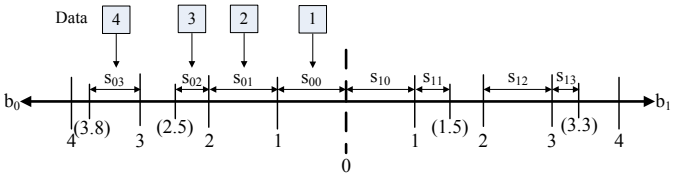


Fig. 5: Initial distribution in HDD bucket, where $data_1$ to $data_4$ are mapped to segments/nodes with pseudo-random generated number sequence.

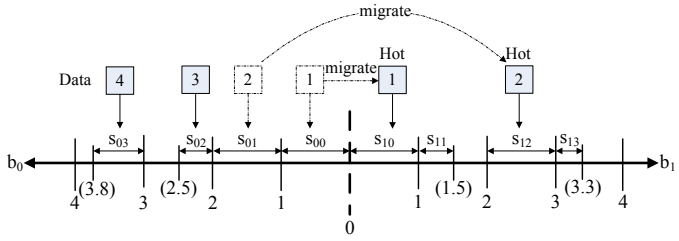
Figure 5 shows an example of the initial distribution, in which four data items with IDs 1 to 4 belong to different segments. With the pseudo-random generators, the random number sequence $\vec{R}$ of each data item is shown as below. The numbers are generated until the hash value matches one segment in the number line of b_0. For instance, when generating the number sequence $\vec{R}$ for $data_1$, the first number 4.2 does not map to any segment range. Then the second number 0.9 is generated, which makes $data_1$ be assigned to s_{00}. Algorithm 1 details the initial data distribution in HDD bucket without replication.

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

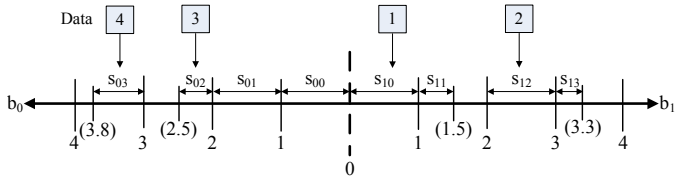
ALGORITHM 1: Initial data distribution in the bottom bucket b_0

Input: data ID x, segment number n, seed e;
$seg[n]$ = segments set
$val \Leftarrow hash(x, e)$
while *x does not belong to any segment* **do**
  $\triangleright generate\ new\ val\ in\ \vec{R}$
  if $val \in\ range\ of\ seg[i]$ **then**
    $seg[i] \Leftarrow x$
  end
end

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. These frequently accessed data will move from b_0 to b_1 when its hotness exceeds SSD bucket threshold v_1. When migrating from the HDD to SSD bucket, the data is mapped to a new segment according to the same random number sequence $\vec{R}$. Figure 6 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 thresholds 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



(a) Data movement from HDD bucket to SSD bucket



(b) Data placement after movement

Fig. 6: Data movement for hot data. $Data_1$ and $data_2$ move from b_0 to b_1, where each data is assigned to a new segment with its original number sequence.

s_{12} ($\vec{r}_0 = 2.7$ for $data2$)), respectively. If no random number matches any segments or the mapped segment/device is worn out prematurely, new numbers will be generated subsequently until they fit one segment. For instance, in the SSD bucket, if the total bytes written (TBW) in one device exceeds a threshold, SUORA will not select the device and continue generating random numbers for the next appropriate one. As such, SUORA tends to move hot data to the bucket with higher performance, and reduces the overhead for recalculation with the same random number sequence $\vec{R}$ when mapping data.

3.3 Hotness Table

As a large-scale storage system maintaining petabytes of data and beyond, it is a non-trivial problem to identify and maintain data hotness efficiently. Numerous methods or functions can be used for hotness computation [30], including the bloom filter [31]. Compared with other methods, such as a flat array search [14], the bloom-filter method has two advantages. First, it can test data set quickly. The insertion and search operations have constant time complexity, based solely on the number of hash functions, unrelated with the data number in the set. Second, it has a low space footprint in implementation. However, one shortcoming for bloom filter is that it may provide a false positive which gives the wrong answer on whether a given key is in the set or not. Fortunately, the false positive can be reduced to a very low level by adjusting parameters (e.g., bloom filter size, the number of hashing functions) [32]. It will not affect the computation of hot data and can be used in our algorithm.

Based on above analysis, we adopt a multi-bloom-filter based technique (we call it *hotness table* in our study) for hotness identification. First, SUORA looks up the hotness table to find the correct bucket for the data by comparing the data hotness value with bucket threshold. Second, it maps the data to the segment with pseudo-random hash functions.

Figure 7 depicts how the hotness table is combined with bucket thresholds in SUORA for hot data identification and data movement. The hotness table uses multiple hash functions (four functions in the example) and multi-bit counters (4-bits but can be larger depending on data scale) for each bit position of the bloom filter. For a data item x, four hash functions generate four positions in the hotness table. Each counter of these four positions is initially set to 0 and is incremented atomically by one to keep data consistent when recording a read data access. All replicas of data x shares the same counters.

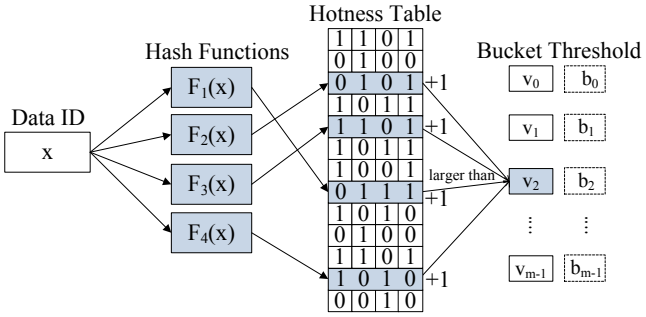


Fig. 7: Hot data identification for a data item by comparing multi-bit counters with bucket threshold, where four hash functions and 4-bit counters are used in the hotness table.

To query data hotness, these four hash functions will generate four positions in the hotness table, but do not increment the counters. Instead, the counts of these four positions are retrieved. If all counts fall into a bucket threshold range v_i to v_{i+1}, which are pre-defined, the data will migrate to bucket i. For example, in Figure 7, all read counters of data x are larger than threshold v_2. Thus data x will migrate from its original bucket to bucket b_2. Depending on the hotness and thresholds, SUORA also allows the data bypass buckets and move to a destination bucket directly.

The hotness table can be implemented and maintained independently in storage nodes with negligible computation overhead. After data is placed on fast devices, the hotness can also be used by the devices for I/O optimizations. For instance, one SSD device can separate hot/cold data to be written in flash memory to improve the performance efficiency of address mapping and garbage collection through its flash translation layer [33].

3.4 Data Migration

3.4.1 Migration for load balance

With the hotness table, SUORA selects the hot data and moves them from slow buckets to fast buckets. The read performance can be improved because these frequently accessed data are placed on fast devices, such as SCMs. The data movement can be performed by periodically comparing data hotness with bucket thresholds. However, one problem is that fast buckets may not have enough space to store the increasing amount of hot data. This is usually true because faster storage devices have smaller capacity. To address it, SUORA will also periodically move data that becomes cold from fast buckets to slow buckets for load/capacity balance.

To reduce the impact on I/O performance, SUORA is designed to trigger cold data migration at idle service time if there is no sufficient space on one bucket. It moves data in batch from the fast bucket to slow bucket. The data migration stops when there is enough free space on the bucket. Otherwise, it would compromise the performance of the heterogeneous storage system since much of capacity in fast buckets remains unused, which conflicts with our design principle discussed before. When the cold data moves, its new bucket position can be decided by the read counter and bucket thresholds.

For each bucket, SUORA uses two watermarks, W_u and W_l, to decide the start and finish time of data migration. The W_u and W_l are upper and lower bound of data occupying on the bucket's total capacity. The optimal watermarks are adjustable in a real storage system depending on the device setting. To void the full

use of space, W_u can be set to a value near to 1, such as 90%. The W_l can be set to make data proportionally distributed on different buckets according to the bucket capacity.

With these two thresholds, SUORA sorts read counters in each bucket and selects a percentage (the value is $W_u - W_l$) of data as the cold data whose hotness counters are in the lower portion. If there are multiple hash functions, SUORA calculates the average value of all read counters for a data item because each hash function has a read counter on the data. During the movement process, SUORA modifies the hotnesss table to reflect the migration of data. When the data in bucket b_i migrates to its previous bucket b_{i-1}, all read counters will be decreased by v_{i-1}.

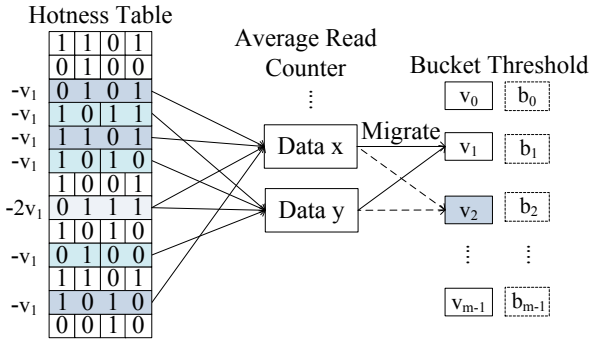


Fig. 8: Update on hotness table for cold data migration by comparing average read counter with bucket threshold.

Figure 8 illustrates the update on hotness table for cold data migration. It can be seen that data x and data y are cold due to their small average read counters. Both of them will move from bucket b_2 to bucket b_1. Simultaneously, all read counters in hotness table will be decreased by the threshold v_1 of bucket b_1. Note that the counter ("0111") in hotness table are decreased twice by v_1 because one hashing value of both data x and y are mapped to the same bit position. The corresponding cold data migration algorithm is described in Algorithm 2. In this manner, SUORA can maximize the performance benefit of SCMs and avoid under-utilization of the capacity of HDDs.

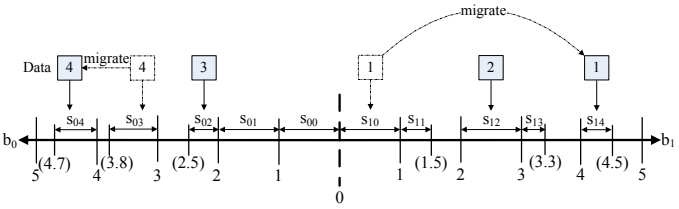
ALGORITHM 2: Cold data migration between buckets

Input: upper watermark W_u, lower watermark W_l, hotness table H_t, bucket number m;

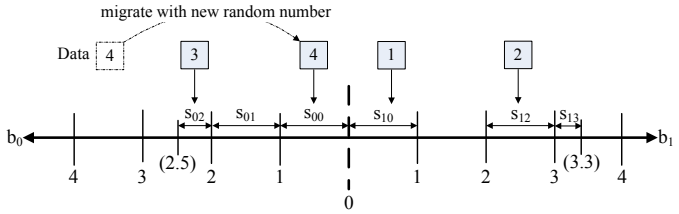
for $i = m-1; i > 0; i--$ **do**
  if $\frac{data\ amount}{total\ capacity\ of\ b_i} > W_u$ **then**
    $\triangleright sort\ data\ in\ b_i\ by\ average\ read\ counter\ in\ an\ ascending\ order$
    while $\frac{data\ amount}{total\ capacity\ of\ b_i} > W_l$ **do**
      $\triangleright get\ a\ data\ item\ in\ order\ from\ the\ sorted\ data\ in\ b_i$
      $move(data, b_i \rightarrow b_{i-1})$
      $H_t \rightarrow decrease(read\ counter(data), v_{i-1})$
    end
  end
end

3.4.2 Migration for node membership change

Besides data movement between different buckets, the data may also migrate when devices (or nodes) are added or removed in the same bucket. This scenario occurs when new storage devices are added into the system or current devices are taken out of the system. For each bucket, data movement only occurs inside it.



(a) Data movement when node addition occurs



(b) Data movement when node removal occurs

Fig. 9: Data movement when node addition and removal occur. In (a), two nodes s_{04} and s_{14} are added, which cause $data4$ and $data1$ to move to them. In (b), two nodes s_{03} and s_{11} are removed, which cause $data4$ to move to a new node.

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 corresponding data will move to the new segment. Otherwise, the data keep its original position. When removing a node, new random numbers are generated for moving data to other nodes.

Figure 9 shows data movement when node addition and removal occurs after the placement described in Figure 6(b). In Figure 9(a), two nodes, s_{04} and s_{14}, are 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. Note that $data1$ and $data4$ are moved 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 9(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 $data4$ in the segment s_{03} moves to the segment s_{00} with a new random number $\vec{r}_3 = 0.8$ as its $\vec{R}$ does not fall into any current segment.

Such an approach achieves appropriate data distribution in accordance with nodes' capacity in a bucket and reduces data movement when nodes are added or removed [15].

3.5 Random Number Functions

As mentioned above, SUORA assigns devices onto segments in each bucket and maps data with pseudo-equally data distribution. 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 [15], as described below.

1. If the data x and seed s are the same, the same random number sequence $\vec{S}$ is generated.
2. If the seed s for the data x is not the same, a different random number sequence $\vec{S}$ is generated.
3. The random numbers in $\vec{S}$ are homogeneously distributed and can be used to map segments in all dimensions.

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 homogeneous distribution in the number line. Suppose the number line of b_2 in Figure 3 is extended from $[0,\ 4)$ to $[0,\ 8)$ and $[0,\ 12)$ for twice. $Data5$ has its initial $\vec{R}1$ and is placed in s_{01} of b_0. When it is moved to b_1, 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.

$\vec{R}1_{data5} = 3.9,\ 1.6\ \in [0,\ 4)$

$\vec{R}2_{data5} = 7.8,\ 1.4,\ 5.8\ \in [0,\ 8)$

$\vec{R}3_{data5} = 11.6,\ 2.3,\ 10.1,\ 3.6,\ 8.2,\ 4.5\ \in [0,\ 12)$

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 $\vec{R}2$ and 3.9 comes from $\vec{R}1$.

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

In 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.

3.6 Replication Algorithm

Replication is a common approach to enhancing data availability in storage systems. When supporting replication, SUORA places replicas in diverse buckets and maps them onto segments/devices via pseudo-random functions. It moves data between HDDs and SCMs to achieve a trade-off among different desired features. In this section, we describe the replication strategy in SUORA, and combine it with data migration to illustrate data distribution on heterogeneous devices.

Suppose there are m buckets with their performance from low to high. Given the replication factor of rep, SUORA will place data on original clockwise rep buckets $\{b_0,\ b_1,\ ...,\ b_{rep-1}\}$, as shown in Figure 3. Each bucket has one copy of stored data, where the data are mapped onto devices with the consideration of capacity and life-time. By access pattern analysis, data movement only occurs if the hotness exceeds the bucket threshold v_{rep}. In each migration, SUORA will move one copy of all hot data from original buckets to new buckets. For the first migration process, SUORA will move hot data from bucket b_{rep-1} to new buckets $b_{rep}, b_{rep+1},\ ..., b_{m-1}$. The new bucket in which to move depends on the data hotness and bucket threshold. One hot data can directly move from bucket b_{rep-1} to bucket b_k if the hotness exceeds the v_k ($rep \leq k < m$). In the second migration process, SUORA will select available replicas of hot data on buckets b_{rep-1} and b_{rep-2}, and move them to new buckets. Consecutively, SUORA will gradually move hot data from original buckets to new buckets.

For instance, given three replicas and five buckets, SUORA will first move one copy of hot data x from bucket b_2 to b_3. If data x is accessed frequently, it will continue increasing the read counter in hotness table (the replicas of one data share same counters). For the next migration, if the hotness of data x exceeds the threshold of b_4, SUORA will directly move the replica of data x on b_1 to b_4. Note that data will not migrate to reduce cost if the target bucket already has the replica of it. With hot data movement, SUORA will make full use of fast devices, while releasing space of slow devices to accommodate replicas of new data. It can also move cold data from fast buckets to slow buckets for load balance. Algorithm 3 shows the pseudo-code of data distribution with replication and migration. The replication consistency is not the focus of this study and can be achieved using existing methods, e.g., a two-phase commit protocol [34].

ALGORITHM 3: Data distribution with replication and migration

```
Input: data set D, bucket number m, segment number n,
       replica number rep, seed e;
seg[m][n] = segment array of buckets
foreach data x ∈ D do
    ▷initial distribution for replication
    for i = 0; i < rep; i++ do
        val ⇐ hash(x, e)
        while x does not belong to any segment do
            ▷generate new val in R⃗
            if val ∈ range of seg[i][j] and
              seg[i][j] is not worn out then
                seg[i][j] ⇐ x
            end
        end
    end
    ▷distribution adjustment for migration
    for i = rep − 1; i ≥ 0; i−− do
        if x has replica on b_i and hotness ≥ v_k then
            move(x, b_i → b_k)
        end
    end
    ▷cold data movement as in Algorithm 2
end
```

The replication algorithm determines the write strategy. For data read, SUORA will retrieve a copy of the data from storage nodes. As mentioned before, SUORA uses a two-step mapping to determine a bucket and the position in the bucket with pseudo-random functions. It reads data from the bucket according to data hotness, which means it prefers reading data from the bucket of higher performance to benefit from SCMs. As the replicas of hot data may move to multiple new buckets, SUORA can request them from different fast devices to significantly reduce read congestion on single device.

3.7 Algorithm Implementation

We have implemented the SUORA algorithm based on Sheepdog [10], a typical, distributed object storage system for virtual machine storage in data centers. It adopts a fully symmetric design, and mainly contains the client (QEMU block driver) and a storage node cluster. When storing an VDI (virtual disk image), the client will divide it into fixed-size objects and send I/O requests to storage nodes. Each storage node can be regarded as a *gateway* to receive client requests, make data mapping, forward requests to target nodes or directly read/write dedicated object files on local file system of nodes.

Specifically, Sheepdog mainly has two types of objects: data object and VDI object. There can be numerous data objects, which contain actual data of virtual disk image. The VDI object is a single object that contains the metadata of virtual disk image, such as image name, disk size, creation time and data object IDs belonging to the image. Relevant VDI objects can be generated for snapshot and cloning operations. As Sheepdog uses consistent hashing to decide data position, the metadata of an VDI object

TABLE 3: Analysis of different algorithms

Algorithm	Computation time		Memory usage	Uniform distribution		Adaptive placement	
	Prepare stage	Distribution stage		Homogeneous	Heterogeneous	Node changes	Hot data
Consistent hashing [8]	$O((uv)\times\log(uv))$	$O(\log(uv))$	$O(uv)$	*fair*	*fair*	*excellent*	*poor*
CRUSH [9]	*negligible*	$O(u)$	$O(u)$	*good*	*fair*	*excellent*	*poor*
ASURA/SPOCA [15, 18]	*negligible*	$O(1)$	$O(u)$	*good*	*fair*	*excellent*	*poor*
SUORA	*negligible*	$O(1)$	$O(u+\epsilon)$	*good*	*excellent*	*excellent*	*excellent*

will not change at most time. It will be updated in cases of image operations, such as creating, deleting, snapshotting and cloning an image (these operations rarely occur). The VDI object can also be updated for new data object creation. If the written data exceeds the size of an object, the client will create a new data object in Sheepdog. At this time, Sheepdog will assign device space for the new data object and update the metadata. It will store the new data object ID to the ID array of the inode structure in a VDI object. The updated metadata will then be written to the VDI object file in target nodes. Sheepdog supports object management and can perform read/write/create/delete operations to objects according to object IDs (similar to simple key-value operations).

For the SUORA algorithm, we use the storage nodes as gateway to collect I/O traces, maintain buckets/segments for devices, and perform data mapping. Each gateway node is modified to have an independent hotness table, using four hash functions and 32-bit counter, to trace read counter for all data requests through the node. When data read or migration happens, atomic operations are achieved for lock on the hotness table to increase or decrease the counter. There is little synchronization overhead between hotness tables. The reason is two-fold. First, multiple virtual machine clients cannot share the same VDI. Second, different VDIs will generate distinct object IDs. As such, the hotness value of objects belonging to one VDI will be identified and kept in the hotness table on the same gateway node once the client connects to it. As multiple storage nodes can be used as gateway, the hotness table will not be a hotspot or prevent scaling in storage system. If one gateway node fails, the client can connect to another gateway node, which will recount the hotness value in its hotness table.

The replicas and data movement are performed periodically in the background to minimize the impact on system performance. From our I/O traces, it shows that there are few VDI object requests compared to data I/O operations. The reason is that Sheepdog uses SUORA (or default consistent hashing) to make data placement, and can directly perform data read/write. Moreover, metadata operations depend on object size, which is set to 64MB to further reduce metadata update overhead. As such, the metadata operations can have little impact on I/O performance.

4 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 [8], CRUSH (straw buckets) [9], and ASURA/SPOCA [15, 18]. We first conducted the evaluation with trace-based simulation, similar to the evaluation mechanism in the CRUSH and ASURA studies. Then we performed tests based on the Sheepdog storage system. We use the original Sheepdog system as the baseline system. We also implemented the CRUSH (straw buckets) algorithm on Sheepdog for performance comparison. Two typical benchmarks, FIO [35] and Filebench [36] are used to generate workloads. The Sheepdog tests were conducted on a local cluster with 30 nodes, which are divided into three 10-node buckets, including two HDD buckets and one SSD bucket. In the HDD buckets, each node has one WD hard disk (500GB WD1200BEVE) or Seagate SATA hard disk (500GB ST9500620NS), respectively. The SSD bucket nodes are equipped with Intel SSDs (200GB SSDSC2BA200G3T). Some device performance can refer to the specification in Table 4, and the segment length is calculated following the equation (1). The Sheepdog storage cluster was connected via 10GbE, and formed with a default replication factor of 2 and object size of 64MB.

4.1 Algorithm Analysis

In this subsection, we evaluate different algorithms based on the analysis from four aspects as described below. Table 3 summarizes the analysis results and algorithm comparison, where the parameter u denotes physical node numbers and v denotes virtual node number (in consistent hashing).

1) **Computation time.** We analyze the algorithm complexity from two aspects: preparation stage and distribution stage. The consistent hashing algorithm calculates the hash values of nodes in the preparation stage, and sorts them (i.e., quicksort) to construct a hash ring. It then calculates the hash value of a data object in the distribution stage, and search for the target node, such as using a binary tree search. Thus, the time complexity of preparation stage and distribution stage for consistent hashing are $O((uv) \times \log(uv))$ and $O(\log(uv))$, respectively. The CRUSH (straw buckets) algorithm calculates the hash values of nodes from data IDs and node IDs. It selects the node that has the largest hash value for the data on the fly, in which the time complexity is $O(u)$. For ASURA and SUORA algorithms, they assign buckets or segments to devices in the preparation stage. The calculation time for assignment is negligible. Both of them achieve nearly $O(1)$ for data distribution because the maximum expectation number of times that random numbers need to be generated to fit a segment depends on a constant value [15]. For SUORA, it maintains the hotnesss number for data placement additionally. The time is negligible because the hotness value is calculated with hash function time, and is compared with a certain number of bucket thresholds at runtime.

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 IDs and their hash values, in which the space complexity is $O(uv)$. The CRUSH (straw buckets) algorithm memorizes u node ID and u node weight, with the space complexity $O(u)$. 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, where each bucket maintains different device and segment information. Besides, SUORA maintains read counters in the hotness table. Thus, the memory requirement of SUORA is $O(u+\epsilon)$, where $O(\epsilon)$ is the memory consumption mainly for hotness table.

3) **Distribution uniformity.** In consistent hashing, the hash values of both nodes and data have variations (i.e., double vari-

ability), which have impact on data distribution in a homogeneous environment. For three other algorithms, they suffer from single variability due to the variation of either hash values or the random numbers. When in a heterogeneous environment, consistent hashing, CRUSH and ASURA can provide a fair data placement according to capacity. Compared to them, the SUORA algorithm can achieve more efficient data distribution with the comprehensive consideration of various device features and data-access patterns.

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

4.2 Compute Time and Memory Footprint

This subsection focuses on understanding the performance of different algorithms. In data distribution, 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) [37], a fast pseudo-random number algorithm, to generate both random numbers and hash values. The simulation was performed in one node with the assumed devices as listed in Table 4, where the average device throughput is measured with a short I/O-intensive test with IOZONE benchmark.

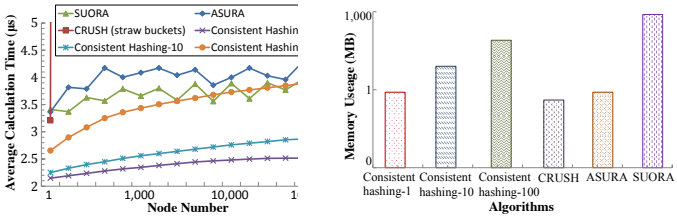
TABLE 4: The specification of devices in the cluster

Device name	Bucket type	Capacity (GB)	Avg. 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
Intel S3700 SSD	b_3	512	500
Intel P3500 SSD	b_4	400	1800

Figure 10 shows the algorithm performance with assuming that the node number varies from 1 to 100,000. Different IDs of data items are generated by the pseudo-random number generator. For ASURA and SUORA algorithms, the nodes 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.

First, we analyze the calculation time of different algorithms. From Figure 10(a), it can be seen that the calculation time of CRUSH (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 spends less time than ASURA as it places all data on half nodes (in the bottom bucket) for data distribution. It reduces the times of random number regeneration, and takes advantage of device characteristics, such as half nodes having larger capacity.

Second, we analyze the memory consumption for data distribution. Suppose there is a total of 10PB data (nearly 0.15 billion data items with 64MB each), as described in Figure 10(b), all algorithms require a low memory footprint less than 700MB with both the node ID and hash number have 4 bytes.



(a) Calculation time (b) Memory consumption

Fig. 10: Calculation time and memory footprint of different algorithms. In (a), the results of SUORA include calculation time of hash values for accessing hotness table. In (b), the vertical axis is in logarithmic scale.

For consistent hashing, it generates virtual nodes so that the memory consumption increases with the addition of virtual nodes. The CRUSH (straw buckets) algorithm maintains $4(u)$ bytes of node ID (assuming with more $1(u)$ bytes of node weight), and needs additional same space when computing the hash value. The ASURA and SUORA algorithms use node IDs and corresponding segment length to place data, which is nearly 0.8MB. For SUORA, it needs additional memory to maintain read counter, which occupies nearly 630MB space given 32-bit counter. In the implementation of SUORA on Sheepdog storage system, the memory for maintaining node IDs and segment length will be resident on each gateway node. For the memory cost of tracing hotness counter, they will be distributed among multiple gateway nodes as each of them has an independent hotness table. Thus, SUORA has a small amount of memory footprint.

4.3 Data Distribution Analysis

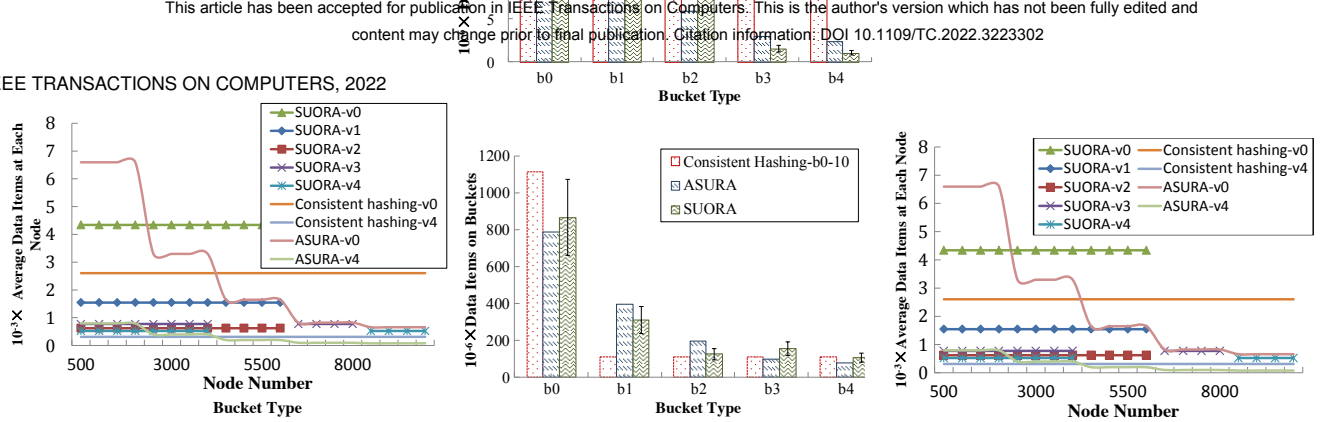
In this subsection, we simulate data distribution uniformity and throughput in a heterogeneous environment. As the calculation time of CRUSH (straw buckets) grows significantly with the increase of nodes, we mainly compare among the other three algorithms.

In these tests, we set the bucket threshold values according to real data-access pattern. To trace the data-access pattern, we deployed the Sheepdog [10] storage system, and traced I/O requests on gateway. Table 5 shows the data-access patterns 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. In our experiment, the nodes are divided into five buckets from b_0 to b_4, where each bucket consists of one type of devices as shown in Table 4. Suppose there is enough bandwidth, the bucket thresholds are set according to the data-access patterns as listed in Table 5. For consistent hashing and ASURA, they do not move data for the consideration of access patterns. Thus, they keep a constant data layout during data placement. For SUORA, it initially distributes data on heterogeneous devices, and then will migrate data between buckets and devices according to hotness.

To understand the statistics, we first formulate the equation for data distribution. Given the total data amount in each type of bucket as $d_0,\ d_1,\ ...,\ d_{m-1}$, the data items on node or segment k (the device is not worn out) in bucket b_i is:

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

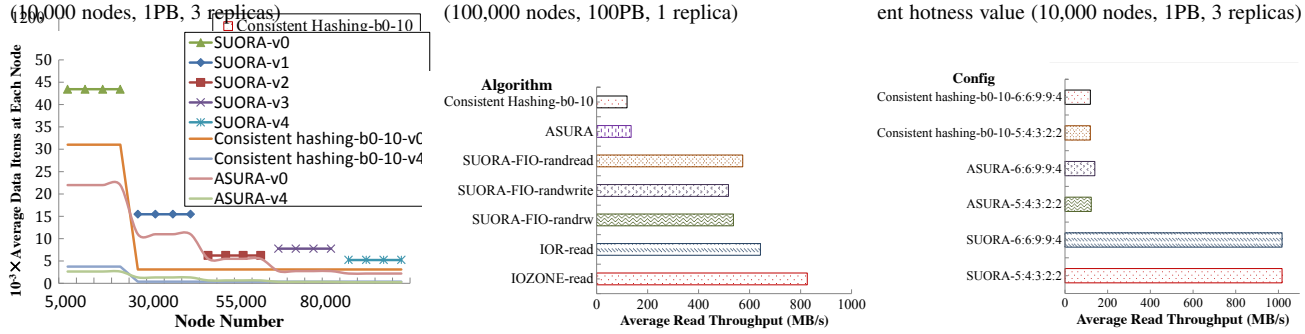
Suppose that each type of bucket has devices with the same average throughput, which is $t_0,\ t_1,\ ...,\ t_{m-1}$. For the data d_i in bucket b_i, different hotness percentages and hotness value are $p_{i0},\ p_{i1},\ ...,\ p_{i(u-1)}$ and $h_{i0},\ h_{i1},\ ...,\ h_{i(u-1)}$, where u is



(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 value (10,000 nodes, 1PB, 3 replicas)



(d) Average data amount at each node with different hotness value (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. 11: Data distribution and throughput for different algorithms with algorithm analysis and simulation

the number of hotness threshold types. The SUORA algorithm can place data according to the hotness and bucket threshold. Consider the various throughput of devices and read times of data, the average read throughput of the storage is:

$$T_{average} = \frac{\sum_{i=0}^{m-1}\sum_{j=0}^{u-1} d_i \times p_{ij} \times h_{ij}}{\sum_{i=0}^{m-1} \frac{\sum_{j=0}^{u-1} d_i \times p_{ij} \times h_{ij}}{t_i}} \quad (3)$$

Figure 11 shows the results of different algorithms, where each data size is set to 64MB. Except in Figure 11(f), every bucket has the same node number. ASURA and consistent hashing algorithms do not distinguish buckets but use the same node setting. Figures 11(a)-(d) show final data distribution of different algorithms. In Figure 11(a), as each data item has 64MB size, the total data amount is about 47 million. For *consistent hashing-1*, it evenly places data on devices in each bucket (nearly 9 million data) but stores excessive data on devices with less capacity. The ASURA algorithm proportionally distributes data among different devices according to the capacity, but does not consider the performance. For SUORA, it initially places data on buckets b_0, b_1 and b_2. As each bucket has one data replica, the data amount on every bucket is nearly 15.5 million. According to the data-access patterns in Table 5 and migration strategy, hot data (nearly 6% of total data amount) will move from original bucket b_2 to new buckets b_3 or b_4. Thus, in the final data distribution, SUORA has different data amount on buckets. Similarly, Figure 11(b) show results in the case of 100,000 nodes and 100PB data. The difference is that for *consistent hashing-b_0-10*, each physical node has 10 virtual nodes in bucket b_0. It means that the data amount on bucket b_0 will be 10 times of that on buckets b_1 to b_4. Since one replica is used, SUORA will migrate more data from bucket b_0 to other buckets because of hotness. Compared with other two algorithms, SUORA can place data on buckets according to device capacity and life-time, while mapping the frequently read data on buckets with higher throughput.

For SUORA, there are few fluctuations for data distribution when access patterns are changed as Table 5 (the series "SUORA" y error bars illustrate changes of data amount moved from slow buckets to fast buckets). The later evaluations on Sheepdog storage system further show the performance efficiency under different workload patterns. To make the algorithm more adaptive to workload changes, the thresholds can be tuned during data placement. For instance, by tracing and analyzing data accesses in previous time window, SUORA can periodically adjust the thresholds for a VDI on the gateway node. As the tuning will make the threshold values change over time, the client may not locate desired data with the data placement decision. SUORA will find the data by searching it among fast buckets in the clockwise direction. Moreover, we can store pre-known hot data in fast buckets according to the hotness and bucket thresholds. It will make full use of SCMs at initial data distribution while avoiding future data movement.

To further understand the algorithm uniformity, we count the final average data amount on each node under *FIO-randread* pattern, as shown in Figure 11(c)-(d). For SUORA, it initially places data among devices in buckets with respect to capacity, as described in Equation (2). In our setting, each bucket consists of one type of devices. Thus, the data will be evenly distributed on devices of bucket. After data migration, the moved data will be placed on new bucket nodes according to equation (2). For original buckets, the remaining data on each device depends on the number of moved data. 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. The similar denotation is used for ASURA and consistent hashing algorithms.

It can be seen that SUORA achieves a more efficient and adaptive distribution compared with others. It distributes most frequently 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 11(c), both b_0, b_1

TABLE 5: Data access patterns

FIO-randread[3]	pct. %	55.58	19.8	7.99	9.93	6.70
	hotness	0[1]	1-25[2]	26-280	281-305	306-659
FIO-randrw	pct. %	65	17.73	8.28	4.76	4.23
	hotness	0	1-18	19-280	281-300	301-792
IOZONE-read	pct. %	64.28	18.29	4.49	7.67	5.27
	hotness	0	1-20	21-50	51-75	76-812
IOR-read	pct. %	63.94	16.33	7.41	8.25	4.07
	hotness	0	1-20	21-100	101-150	151-819

[1] 0 means 55.58% data are not accessed.
[2] 1-25 means there is 19.8% data with the read number between 1 and 2
which is also used as setting for bucket threshold values in this pattern.
[3] In FIO-randread trace, the bucket thresholds of $\{b_0,\ b_1,\ b_2,\ b_3,\ b_4\}$ ca
be set to $\{0,\ 1,\ 26,\ 281,\ 306\}$.

and b_2 have data with different read frequency. It is because th
SUORA places replicas on them and only migrates data from
to b_3 or b_4 until the hotness exceeds v_3. This significantly reduc
the data movement amount. In contrast, consistent hashing even
places the data among all devices regardless of hotness. Althou
there is a little fluctuation in data placement for frequency, the
ASURA algorithm lacks an effective method to distinguish differ-
ent devices.

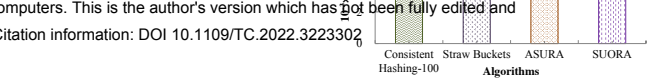
Figure 11(e)-(f) show the average read throughput under different patterns and configurations. Obviously, the average performance is related with the throughput of each bucket. In Figure 11(e), it can be seen that SUORA achieves the best performance. The average read throughput of SUORA is nearly improved from 3.9 to 8.5 times compared to consistent hashing and ASURA algorithms. This is because that SUORA 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 data patterns. For them, the performance is mainly affected by virtual node numbers and device capacity, respectively. Figure 11(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.

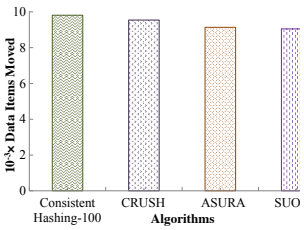
4.4 Data Migration Evaluation

In this subsection, we simulate the data migration of different algorithms. For SUORA, data movement can occur in one bucket for node addition and removal, or between buckets due to the change of hotness. We conducted two sets of tests and analyzed the cost of data movement.

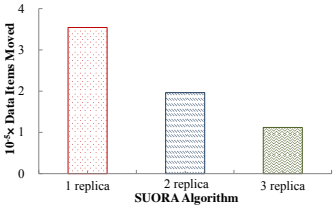
For the first set of tests, we use the setting in bucket b_0, where we assume to have 100 nodes and 1 million data items. Figure 12(a) shows the data movement when a new node is added for different algorithms. We calculate the total number of data that will move to the newly added node, as seen in the vertical axis. These tests indicate that all algorithms achieved similar results. Consistent hashing and CRUSH (straw buckets) algorithms evenly distribute data across all nodes, and they can achieve small data movement when node changes. Suppose there are u nodes, nearly $1/(u+1)$ data will move to the newly added node. Compared with these two algorithms, SUORA and ASURA use pseudo-random functions to distribute data on nodes. They achieved small data movement too. Similar results can be observed when a node is removed.

For the second set of tests, we evaluate data movement between buckets in SUORA. We use five buckets (from b_0 to b_4,





(a) Data movement when a node is added



(b) Data movement using different replicas

Fig. 12: The cost of data movement for different algorithms

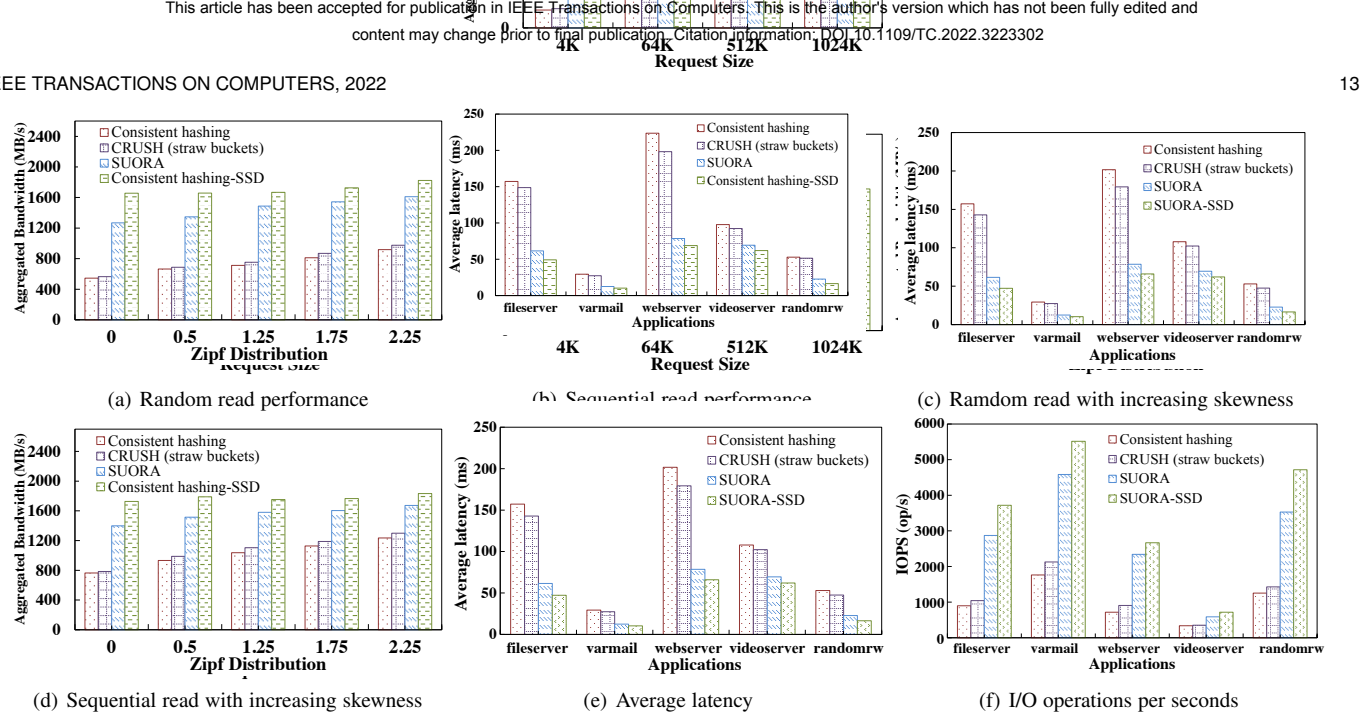
the same as in Section 4.3) and 1 million data items for initial data distribution. We calculate the total number of data that will move between buckets under the IOR-read pattern. Figure 12(b) shows the results using different replicas. It can be seen that there is a large amount of data movement with one replica, which is about 35% of the total amount of data. The high cost is because all data are initially placed in the b_0 bucket and will move to other buckets when they are accessed more often. On the contrary, with the increase of the number of replicas, there are significant reduction of data movement cost. For instance, when using three replicas, the amount of data movement is less than 12.5% of the total amount of data, which is nearly one third of the case with one replica. The significant reduction of data movement is because SUORA does not migrate data to a bucket if the target bucket already has a copy of it. These tests verified that SUORA can use replication strategy to achieve efficient data movement between buckets with a small amount of cost.

4.5 FIO Performance

We further tested the read performance of different algorithms with FIO benchmark on the 30-node Sheepdog storage system. We used multiple virtual machine clients on different nodes to launch FIO benchmark. For SUORA algorithm, the nodes are divided into three buckets, including two HDD buckets and one SSD bucket. More specifically, their thresholds are set as $\{0,\ 1,\ 30\}$ according to the statistics in Table 5. SUORA initially places data on HDD buckets, where each bucket has one copy of stored data. The data distribution among devices in bucket complies with equation (2). According to the data-access patterns, nearly 20% of total data moved from HDD buckets to SSD bucket at the end of benchmark.

Figure 13(a) and (b) report the FIO test results of different algorithms. For comparison, we also test the I/O performance of Sheepdog by only using 10 SSD nodes as storage nodes, namely *consistent hashing-SSD*. We run 8 clients on different nodes, in which each client launches one FIO instance with 8 jobs and request sizes from $4KB$ to $1MB$. The bandwidth was aggregated by adding each client. As seen from the figures, the performance of all algorithms increases with the increase of request size. Consistent hashing and CRUSH achieved the similar bandwidth because both of them distribute data evenly on different nodes. Compared to consistent hashing, SUORA and consistent hashing-SSD achieved better bandwidth by 112% to 246%. The reason is that they can take advantage of the performance benefit of SSDs. For SUORA, it can move hot data to fast buckets, and access the data replica from the SSD devices. The performance advantage of SUORA is as expected. These tests and observations confirm the efficiency of the SUORA algorithm in a heterogeneous storage system.

The SUORA algorithm uses hotness table to detect hot data and access them from fast buckets. To validate the effectiveness, we use unbalanced data accesses as the workload for evaluation.



(a) Random read performance (b) Sequential read performance (c) Ramdom read with increasing skewness

(d) Sequential read with increasing skewness (e) Average latency (f) I/O operations per seconds

Fig. 13: I/O Performance comparison for algorithms under different workloads and applications. (a)-(d) are FIO read performance with different request sizes or skewness workloads; (e) and (f) are file system application performance with mixed read/write workloads.

We run FIO to measure read operations with increasingly skewed data access distribution, namely *Zipf* distribution [38]. The FIO benchmark generates more skewed data distribution with the increase of Zipf parameter, meaning that a proportion of data are more frequently accessed than others.

Figure 13(c) and (d) show the read performance with $512KB$ request size in different Zipf distributions. It can be observed that both SUORA and consistent hashing-SSD achieved better performance. Moreover, SUORA has higher bandwidth with increasing skewness. This is because when more data become hot, SUORA has more chances to access the hot data replicas on fast buckets. Thus the performance of SUORA in a heterogeneous cluster is close to consistent hashing-SSD. On the other hand, consistent hashing evenly distributes data among different storage nodes. Although there is slow performance increase with larger skewness, consistent hashing can not leverage the benefits of SSDs well.

4.6 File System Workload Evaluation

For performance evaluation, we also conduct tests with the Filebench benchmark, which emulates file system level workloads of different real applications. We selected five types of applications in Filebench with the specification described in Table 6, where *WF* means reading or writing a whole file. From *fileserver* to *videoserver* applications, we emulated sequential access to files (or append to files). For the *randomrw* application, we emulated random access to files (includes partial write). We generated various data sets with repeated accesses, and set read/write ratio from 1:1 to 10:1. For the watermarks, we set W_u and W_l to 20% and 10% to trigger cold data movement. We launched one client running on one virtual machine on the Sheepdog.

TABLE 6: The specification of emulated file system workloads

Application	Average File Size	I/O size	Data Access
fileserver	256KB	WF	sequential
varmail	32KB	WF	sequential
webserver	256KB	WF	sequential
videoserver	1GB	1MB	sequential
randomrw	128KB	64KB	random

Figure 13(e) and (f) report the performance results of different algorithms under five workloads. We also test SUORA-SSD for comparison, which only uses the SSD bucket nodes for storage. As seen from these figures, SUORA achieved close performance to SUORA-SSD, in which both of them behaved better than other algorithms. This is because SUORA can distinguish the performance difference between slow and fast buckets. It will move data among buckets and access hot data from fast buckets, such as SSDs. SUORA can also improve the entire I/O performance while maintaining load balance for data movement by considering device capacity. In contrast, the consistent hashing and CRUSH algorithms evenly placed data among different devices without being aware of their distinct characteristics. Additionally, SUORA favored high performance like SUORA-SSD on the *randomrw* application, which accessed data with small and random I/O sizes. The results further confirm the efficacy of the SUORA algorithm in a heterogeneous environment.

For write traffic, the gateway in Sheepdog calculates the target nodes with distribution algorithm, and sends write requests to all of the target nodes. As the virtual machines cannot share the same VDI at the same time, there is no write-write conflicts. The write optimization is often more complicated in storage systems due to object cache layer, replica consistency, asynchronous I/O behaivor, etc. In our evaluation, it was also observed that SUORA achieved better access performance over the existing replication schemes due to the fact that it well considers distinct throughput of heterogeneous devices.

4.7 System Overhead

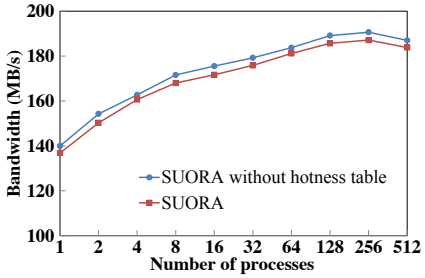


Fig. 14: System overhead for sequential read

The SUORA algorithm uses hotness table to identify hot data and locate the data on buckets. When reading data, it maintains

read counters and compare the counters with bucket thresholds. Besides this, SUORA updates the hotness table for cold data movement. To measure the overhead caused by these operations, we evaluated the impact of hotness table in SUORA.

Figure 14 shows the sequential read performance with one client running on Sheepdog, which uses two HDD buckets and one SSD bucket as storage nodes. The case of *SUORA without hotness table* means the hotness table is disabled so that the data are directly stored in the bucket. The results show that the overhead of maintaining hotness table is very minor and negligible.

5 Conclusions

The adoption of new non-volatile memory devices has introduced significant challenges for data management in large-scale parallel/distributed file systems. In this research, we propose a novel data distribution algorithm called SUORA, which considers distinct device features (capacity, performance and life-time) and data-access patterns (hotness and learned thresholds) to uniformly place data cross a hybrid and tiered storage cluster. It combines data replication with migration to make full use of storage-class memory devices for performance efficiency, while maintaining load balance and very small movement cost. Evaluation results show that SUORA achieved highly efficient data mapping and largely improved the I/O bandwidth compared against other data distribution algorithms in a heterogeneous environment.

Acknowledgments

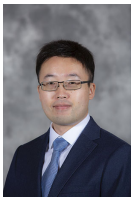
This research is supported in part by the National Science Foundation under grant CCF-1718336, CCF-1853714 and CNS-1817094.

References

[1] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *Proc. FAST Conf.*, 2002, pp. 231–244.
[2] S. A. Weil, S. A. Brandt, E. L. Miller, and D. D. E. Long, "Ceph: A scalable, high-performance distributed file system," in *Proc. USENIX Oper. Syst. Des. Implementation*, 2006, pp. 307–320.
[3] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," in *Proc. ACM Symp. Oper. Syst. Principles*, 2003, pp. 29–43.
[4] H. Wong, S. Raoux, S. Kim, J. Liang, J. Reifenberg, B. Rajendran, M. Asheghi, and K. Goodson, "Phase change memory," *Proc. IEEE*, vol. 98, pp. 2201–2227, 2010.
[5] Z. Diao, Z. Li, S. Wang, Y. Ding, A. Panchula, E. Chen, L. Wang, and Y. Huai, "Spin-transfer torque switching in magnetic tunnel junctions and spin-transfer torque random access memory," *Journal of Physics: Condensed Matter*, vol. 16, no. 19, 2007.
[6] "Intel and Micron produce breakthrough memory technology," 2015. [Online]. Available: https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/
[7] S. He and X. Sun, "A cost-effective distribution-aware data replication scheme for parallel I/O systems," *IEEE Trans. Comput.*, vol. 67, no. 10, 2018.
[8] 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. Annual ACM Symp. Theory Comput.*, 1997.
[9] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proc. ACM/IEEE Conf. Supercomputing*, 2006, pp. 654–663.
[10] "Sheepdog Project," 2018. [Online]. Available: https://sheepdog.github.io/sheepdog/
[11] "Glusterfs file system project," 2018. [Online]. Available: http://www.gluster.org/
[12] S. He, X. Sun, and A. Haider, "HAS: Heterogeneity-aware selective layout scheme for parallel file systems on hybrid servers," in *Proc. Int. Paralel Distrib. Process. Symp.*, 2015.
[13] S. Ma, H. Chen, Y. Shen, H. Lu, B. Wei, and P. He, "Providing hybrid block storage for virtual machines using object-based storage," in *Proc. IEEE ICPDS Conf.*, 2014, pp. 150–157.
[14] J. W. Hsieh, L. P. Change, and T. W. Kuo, "Efficient identification of hot data for flash memory storage systems," *ACM Trans. Storage*, vol. 2, no. 1, pp. 22–40, 2006.
[15] K. I. Ishikawa, "ASURA: Scalable and uniform data distribution algorithm for storage clusters," *arXiv preprint arXiv:1309.7720*, 2013.
[16] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *Proc. MSST Conf.*, 2010, pp. 1–10.
[17] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *ACM SIGOPS Oper. Syst. Review*, vol. 44, no. 2, pp. 35–40, 2010.
[18] A. Chawla, B. Reed, K. Juhnke, and G. Syed, "Semantics of caching with SPOCA: A stateless, proportional, optimally-consistent addressing algorithm," in *Proc. USENIX ATC Conf.*, 2011.
[19] J. Lamping and E. Veach, "A fast, minimal memory, consistent hash algorithm," in *arXiv preprint arXiv:1406.2294*, 2014.
[20] M. Matsumoto and T. Nichimura, "Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator," *ACM Trans. Model. Comput. Simul.*, vol. 8, no. 1, pp. 3–30, 1998.
[21] B. Welch and G. Noer, "Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions," in *Proc. MSST Conf.*, 2013.
[22] J. Zhou, W. Xie, Q. Gu, and Y. Chen, "Hierarchical consistent hashing for heterogeneous object-based storage," in *Proc. ISPA Conf.*, 2016.
[23] E. Kakoulli and H. Herodotou, "OctopusFS: A distributed file system with tiered storage management," in *Proc. ACM Int. Conf. Manage. Data*, 2017.
[24] S. He, Y. Wang, X. Sun, and C. Xu, "HARL: Optimizing parallel file systems with heterogeneity-aware region-level data layout," *IEEE Trans. Comput.*, vol. 66, no. 6, 2017.
[25] S. He, Z. Li, J. Zhou, Y. Yin, X. Xu, Y. Chen, and X. Sun, "A holistic heterogeneity-aware data placement scheme for hybrid parallel I/O systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 4, 2020.
[26] J. Zhou, Y. Chen, W. Xie, D. Dai, S. He, and W. Wang, "Prs: A pattern-directed replication scheme for heterogeneous object-based storage," *IEEE Trans. Comput.*, vol. 69, no. 4, 2019.
[27] J. Zhou, W. Xie, J. Noble, K. Echo, and Y. Chen, "SUORA: A scalable and uniform data distribution algorithm for heterogeneous storage systems," in *Proc. NAS Conf.*, 2016.
[28] K. Ganesh, Y. Kim, M. Debnath, S. Park, and J. Lee, "LAWC: Optimizing write cache using layout-aware I/O scheduling for all flash storage," *IEEE Trans. Comput.*, 2017.
[29] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely, and J. Emer, "High performing cache hierarchies for server workloads: Relaxing inclusion to capture the latency benefits of exclusive caches," in *Proc. HPCA Conf.*, 2015.
[30] Q. Li, L. Shi, C. Gao, Y. Di, and C. Xue, "Access characteristic guided read and write regulation on flash based storage systems," *IEEE Trans. Comput.*, vol. 67, no. 12, 2018.
[31] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
[32] D. Park and D. Du, "Hot data identification for flash-based storage systems using multiple bloom filters," in *Proc. MSST Conf.*, 2011.
[33] W. Xie, Y. Chen, and P. C. Roth, "Exploiting internal parallelism for address translation in solid-state drives," *ACM Trans. Storage*, vol. 14, no. 4, pp. 1–30, 2018.
[34] M. T. Ozsu and P. Valduriez, "Principles of distributed database systems," Springer Science and Business Media, Tech. Rep., 2011.
[35] "The FIO Tool Benchmark," 2015. [Online]. Available: http://freecode.com/projects/fio
[36] "The File System Benchmark," 2018. [Online]. Available: http://sourceforge.net/projects/filebench
[37] "Simd-oriented fast mersenne twister: a 128-bit pseudorandom number generator," 2017. [Online]. Available: http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html
[38] "Zipf Distribution," 2017. [Online]. Available: https://en.wikipedia.org/wiki/Zipf's_law.

Jiang Zhou is an Associate Professor in the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include file and storage systems, parallel and distributed computing, metadata management, I/O optimization, and cloud computing.

Yong Chen is an Associate Professor and Director of the Data-Intensive Scalable Computing Laboratory in the Computer Science Department of Texas Tech University. His research interests include data-intensive computing, parallel and distributed computing, high-performance computing, and cloud computing.



Mai Zheng is an Assistant Professor at Iowa State University. His research interests include file systems, non-volatile memories, key-value stores, data infrastructures, data-intensive computing.

Weiping Wang received the Ph.D. degree in computer science from Harbin Institute of Technology, China, in 2008. He is a Professor in the Institute of Information Engineering, Chinese Academy of Sciences. His research interests include database and storage systems.