Design of Global Data Deduplication for A Scale-out Distributed Storage System

Myoungwon Oh*§, Sejin Park† , Jungyeon Yoon*, Sangjae Kim*, Kang-won Lee*,
Sage Weil¶, Heon Y. Yeom§, Myoungsoo Jung‖
* SK Telecom, †Keimyoung University, ¶ Red Hat, § Seoul National University, ‖Yonsei University
Email: {omwmw,jungyeon.yoon,sangjae,kangwon}@sk.com, baksejin@kmu.ac.kr
sweil@redhat, yeom@snu.ac.kr, mj@camelab.org

***Abstract*—Scale-out distributed storage systems can uphold balanced data growth in terms of capacity and performance on an on-demand basis. However, it is a challenge to store and manage large sets of contents being generated by the explosion of data. One of the promising solutions to mitigate big data issues is data deduplication, which removes redundant data across many nodes of the storage system. Nevertheless, it is non-trivial to apply a conventional deduplication design to the scale-out storage due to the following root causes. First, chunk-lookup for deduplication is not as scalable and extendable as the underlying storage system supports. Second, managing the metadata associated to deduplication requires a huge amount of design and implementation modifications of the existing distributed storage system. Lastly, the data processing and additional I/O traffic imposed by deduplication can significantly degrade performance of the scale-out storage.**

To address these challenges, we propose a new deduplication method, which is highly scalable and compatible with the existing scale-out storage. Specifically, our deduplication method employs a double hashing algorithm that leverages hashes used by the underlying scale-out storage, which addresses the limits of current fingerprint hashing. In addition, our design integrates the meta-information of file system and deduplication into a single object, and it controls the deduplication ratio at online by being aware of system demands based on post-processing. We implemented the proposed deduplication method on an open source scale-out storage. The experimental results show that our design can save more than 90% of the total amount of storage space, under the execution of diverse standard storage workloads, while offering the same or similar performance, compared to the conventional scale-out storage.

1. Introduction

Data is exploding and generated in just the last year is expected to be more than that in the entire previous of industry. To store this massive volume of data more efficiently, both scalable storage and a data reduction technique such as deduplication are needed. Therefore, the scale-out storage such as GlusterFS [16] and Ceph [18] is exploited in diverse computing domains, ranging from a small cluster to distributed system and to high performance computing, because they can expand their capacity on demand. Also,

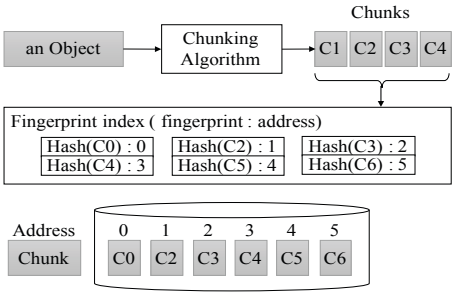


Figure 1: Traditional data deduplication system.

many previous studies [7] [8] [31] [34] demonstrated that data deduplication can reduce lots of redundant data with a low overhead, which makes applying data deduplication on distributed storage systems promising to maximize storage capacity of diverse systems. However, simply applying deduplication is inappropriate for distributed storage environments because it removes all extra copies appended by redundancy scheme. In a distributed environment, employing redundancy scheme such as replication and erasure coding [2] [3] [4] is required to achieve a high availability against frequent failures of the underlying storage. Therefore, a design of global data deduplication that not only removes the redundant data but also preserves underlying redundancy scheme residing in the existing scale-out storage system is required.

However, applying data deduplication on the existing distributed storage system considering redundancy scheme is non-trivial due to its data processing procedure and additional metadata management. Figure 1 shows an example of data processing observed in a traditional data deduplication method [5] [6]. In this example, the input source as a data object is split into multiple chunks by a chunking algorithm. The deduplication system then compares each chunk with the existing data chunks, stored in the storage previously. To this end, a fingerprint index that stores the hash value of each chunk is employed by the deduplication system in order to easily find the existing chunks by comparing hash value rather than searching all contents that reside in the underlying storage. Even though aforementioned deduplication process is straightforward to be implemented for many systems, it does not work well with the existing scale-out storage systems because of the following reasons. First, managing scalability of fingerprint index can be problematic with a limited working memory size. Specifically, the fingerprint index is one of the most essential and expen-

- *Myoungwon Oh**§ is with Seoul National University.*

2575-8411/18/$31.00 ©2018 IEEE
DOI 10.1109/ICDCS.2018.00106

sive components in modern data deduplication systems [12] [5] [33] [37]. As storage capacity grows, the size of the fingerprint index increases proportionally. In other words, the fingerprint index cannot be stored in working memory. Note that for fast comparison, a fingerprint index should be located in memory. In addition, there are more problems regarding the management of such fingerprint index. If the underlying storage is a shared-nothing distributed system that has no centralized metadata server (MDS), there is no explicit space to locate the fingerprint index. Second, it is complex to ensure compatibility between newly applied deduplication metadata and existing metadata. To apply deduplication metadata such as reference counts and fingerprint indexes, a new solution that incorporate them with existing storage system is required. Third, we need to solve performance degradation due to additional operations by adding deduplication. Deduplication requires a fingerprint operation and an additional I/O operation on the storage system, therefore, performance can be degraded. Especially, if foreground I/O is affected by background deduplication, serious performance degradation could occur. Also, deduplication of frequently used data will result in unnecessary I/O and eventual performance degradation.

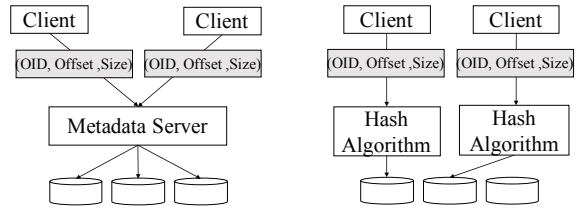
In this paper, we propose a unified data deduplication design with the following characteristics. We effectively remove the fingerprint index by leveraging underlying storage system's existing hash table (*Double hashing*), and extend its metadata structure to effectively support data deduplication (*Self-contained object*). Also, performance degradation is minimized through rate control and selective deduplication based on post-processing. We implemented the proposed design upon open source distributed storage system, Ceph [18].

There are a few studies on data deduplication to apply it into the existing distributed (or clustered) storage systems [9] [10] [11]. Unfortunately, they depend on MDS, which keeps metadata independently for management and storage thereby introducing multiple single point of failure (SPOF). Even though HYDRAstor [35] has no dependency with MDS, its design cannot easily integrated into the existing scale-out storage that supports a high availability, data recovery, and data re-balancing. The main reason behind this challenge is that its data structures for data deduplication are separated from data store. Therefore, such storage features for the data store cannot cover the additional data structures of data deduplication such as fingerprint index and reference count.

The main contributions of this paper can be summarized as follows:

(1) An effective deduplication approach to support high availability and high performance in distributed storage system: we provide deduplication design that is applicable on replication or erasure coding and minimize performance degradation caused by deduplication.

(2) A unified data deduplication design for distributed storage system: the proposed design does not require an additional fingerprint index or data structure for deduplication without any modification of existing storage structure.



(a) An ordinary MDS-based distributed storage (Centralized).

(b) An ordinary HT-based distributed storage (Decentralized).

Figure 2: Overview of distributed storage systems.

(3) Minor changes for the existing distributed storage systems: for distributed hash table based scale-out storage system, only a few code modification is required to support data deduplication. In our implementation on Ceph, only about 4K of lines are added or modified.

2. Background

2.1. Target Distributed Scale-out Storage System

Distributed scale-out storage systems can be classified based on how they share information: Centralized or decentralized (shared-nothing). In the centralized storage, a metadata server (MDS) stores connecting information between a data and a storage and in the decentralized storage, a hash algorithm determines the placement of a data. Figure 2 depicts these two types of distributed storage system. All the client requests (e.g., object ID, offset and size) are translated by an MDS or a hash algorithm. Specifically, Figure 2-(a) shows the organization of an ordinary MDS-based distributed storage system. All requests, targeting to the storage server need to pass to the MDS in obtaining metadata, which in turn makes MDS performance an important factor for the whole storage system. In addition, it requires to provide the availability of MDS in order to cope with failure case. Lustre [14], GFS [13], and pNFS [15] are implemented by this MDS based design.

On the other hands, this design should consider a single point of failure (SPOF) problem and scalability of MDS. Figure 2-(b) shows an ordinary hash table based distributed storage system, which employs the shared-nothing structure and decentralizes given objects without MDS. Incoming requests are constantly translated by the hash table. Thanks to their superiority in terms of scalability and availability, many recent scale-out storage systems such as Ceph [18], GlusterFS [16], Swift [17], Dynamo [19] and Scality [20] are developed based on the decentralized structure. In Ceph, CRUSH algorithm is used for object distribution and GlusterFS uses distributed hash table (DHT) based algorithm to distribute objects efficiently. In this paper, we aimed at a decentralized shared nothing storage system for the higher scalability and availability.

2.2. Deduplication Range

Applying deduplication to the existing distributed storage systems is complex because we need to deduplicate all data while keeping the rules of the existing systems. One of

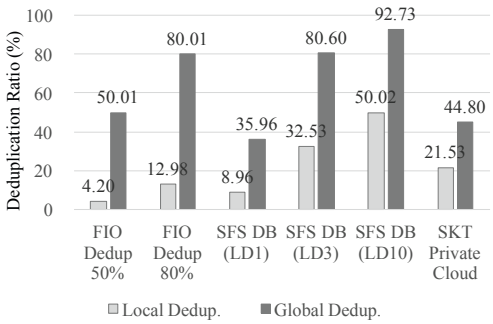


Figure 3: Deduplication ratio comparison between global deduplication and local deduplication. For this experiment, 4 Ceph storage nodes are used and each node has 4 OSDs (Object Storage Device). Local deduplication targets per-OSD basis and global deduplication targets all 16 OSDs.

	4 OSD	8 OSD	12 OSD	16 OSD
Local Dedup.	15.5 %	8.1 %	5.5 %	4.1 %
Global Dedup.	50.0 %	50.0 %	50.0 %	50.0 %

TABLE 1: Deduplication ratio comparison between global deduplication and local deduplication depends on the number of OSD. Workload is a FIO workload with deduplication ratio of 50%.

the most straightforward ways that can deduplicate without any violation against the policies of underlying storage system is to individually apply the deduplication to each single node by a leveraging block level deduplication solution such as [29] [28]. The most compelling advantage of this local deduplication is that distributed storage system's various policies can be consistently maintained because most of them operate on per-node basis: data replication, erasure coding, and data balancing. However, the local deduplication suffers from a limited deduplication ratio compared to global deduplication.

Figure 3 compares the deduplication ratio of local deduplication and global deduplication by executing micro benchmarks and real workloads that SK maintains for private cloud. Specifically, it includes two FIO [38] workloads with different deduplication ratio (size: 5GB), three SPEC SFS 2014 [39] database workloads with different loads (size: 24GB) and a real word workload of enterprise cloud data in our private cloud (size: 3.3TB). Our private cloud has about 100 VMs for developers ranging from 50GB to 500GB, and the data excluding OS images is over-provisioned in Ceph. More detailed experimental environment is described in Section 6. In the first two workloads, which are artificially formed by FIO, the local deduplication shows severely low deduplication ratio while global deduplication shows the same results as given deduplication ratios. In the SFS DB workloads, the local deduplication still shows two to four times lower deduplication ratio than that of global deduplication based on given loads. With our private cloud workloads, the difference of deduplication ratio between the two methods is still around two times. This result shows that the limited deduplication ratio problem of local deduplication has a negative effect not only on synthetic workload but also on real environment. In addition, as the number of OSD increases, the gap between local deduplication and global deduplication increases, as shown in Table 1. This illustrates that the larger the size of the distributed system, the more significant the problem of limited deduplication ratio of local deduplication becomes. In this paper, we target global deduplication for a distributed storage system to achieve a higher deduplication ratio.

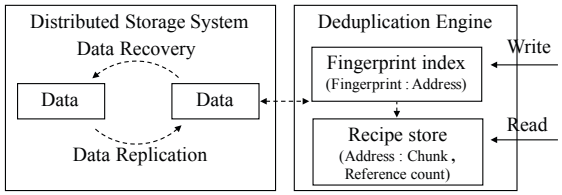


Figure 4: Common approach to support data deduplication.

3. Problem and Key Idea

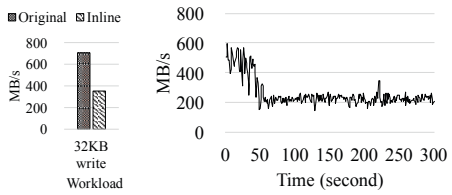
3.1. Problem Definition

In this Section, we will define practical problems to apply deduplication on distributed storage system.

Scalability of fingerprint index: There are two challenges to manage scalability of fingerprint index: how to lookup fast and how to distribute evenly. First, the increased fingerprint index makes fast memory lookup difficult. For example, if an entry of fingerprint index needs at least 32 bytes, not only huge storage capacity is needed but also fingerprint index cannot reside in memory when storage capacity reaches more than hundreds of PB. Therefore, fingerprint-lookup degrades overall performance. Many previous studies [12] [33] [37] proposed representative fingerprint techniques to reduce the size of index table. Representative fingerprint based approaches have an advantage of small index table, but they cannot remove all of the duplicated chunks. Moreover, in distribute storage system, theoretically, fingerprint index will grow unlimitedly even though their policy is maintaining a small set.

Another challenge is how to locate and distribute the fingerprint index equally. The existing method is a centralized metadata server (MDS) [9] [10] [11]. However, if MDS is used, it causes the other problems: SPOF and performance bottleneck. Because of these problems, a new distribution method for deduplication metadata that is suitable for decentralized scale-out systems is needed.

Compatibility between the newly applied deduplication metadata and exiting metadata: Most previous works have external metadata structure for deduplication metadata since this design can be implemented relatively easily [5] [35] [10]. However, additional complex linking between deduplication metadata and existing scale-out storage system is required. Figure 4 shows a conceptual diagram of common approach to support data deduplication. For example, if we add fingerprint index or reference count information for deduplication, the underlying system's storage features do not recognize these additional data structure. Therefore, the storage features such as high availability or data recovery cannot work for the external data structure. These features must be implemented separately in the external data structure. Even worse, it is hard to guarantee that



(a) Partial write problem. (inline processing) (b) Interference problem. (post processing)

Figure 5: Performance degradation brought by the existing deduplication method.

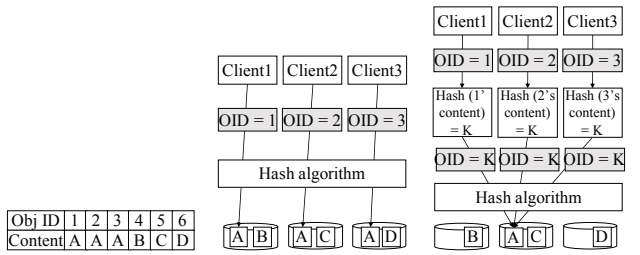
the modified or added module works correctly or not. The correctness for high availability is critical for reliability.

Minimizing performance degradation: There are also two challenges to minimize performance degradation on applying deduplication. First, when to start deduplication needs to be considered. The methods of a deduplication can be also classified based on deduplication timing: inline and post-processing. Inline deduplication removes the redundant data immediately, while introducing additional deduplication latency at runtime. Although this approach is slower than post-processing, it has the advantage that no additional spcae for temporal store is required [25]. In contrast, the post-processing deduplication conducts deduplication in background process, which exhibits a better performance than inline. However, a foreground I/O request can be interfered by the background deduplication process. The both approaches have their own pros and cons respectively. Therefore, it is necessary to devise what is the ideal solution for distributed system. For better understanding, we configured the same experimental environment used in the evaluation section (cf. Section 5) and conducted the experiment by issuing a sequential write (foreground I/O) while inline processing is enabled or a background deduplication thread operates. We observe the performance problem of each deduplication method. Figure 5-(a) illustrates the partial write problem of inline processing. In this experiment, a foreground I/O service is issued using 16KB block size while the chunk size is 32KB. Therefore, the I/O service can be finished after its read-modify-write is completed (reading 32KB chunk $\Rightarrow$ modifying 16KB data $\Rightarrow$ writing 32KB chunk). This causes significant performance degradation. Figure 5-(b) depicts foreground I/O interference problem. The throughput of the foreground I/O slows from 600MB/s to 200MB/s.

Second, how to deal with hot data needs to be considered. Applying deduplication to frequently used data causes the overhead because it will be updated soon again. If deduplication is continuously applied on hot data during I/O, unnecessary frequent I/O for deduplication will cause performance degradation.

3.2. Key Idea

Double hashing: To solve the fingerprint index problem, we propose a *Double hashing* mechanism. The key mechanism of fingerprint index is to detect redundant chunks faster. In a traditional deduplication, a hash value of a chunk and a location of the chunk are mapped in the fingerprint



(a) ObjID – content relation. (b) An ordinary OID-based distributed Storage. (c) A content-hashed OID-based distributed Storage.

Figure 6: An ordinary hashing in distributed storage system and *Double hashing* in the proposed system.

index. So, when we query a new chunk's hash value to the fingerprint index, we can retrieve its location. Interestingly, a distributed storage system has similar mechanism [5]. A distributed hash algorithm determines the location of an object. One difference is the input value for the hash algorithm. Namely, a hash value of a chunk is an input for a deduplication system and object ID is an input for a distributed storage system.

The key idea of the *Double hashing* is to combine these two mismatched input value. Figure 6 depicts this idea. Figure 6-(a) shows a relationship between an object ID of data and its contents. In this example, object ID 1, 2, 3 have the same contents. Figure 6-(b) shows an ordinary distributed storage system addressed by object IDs. Each client can find an object data for an object ID using predefined hash algorithm. e.g. CRUSH algorithm in Ceph and DHT algorithm in GlusterFS. However, since there is no relationship between content and object ID, the same data could exist across multiple storage nodes. In this situation, to find the same contents for deduplication, traversing all the storage node or maintaining a big fingerprint index is required. We remap the ordinary policy-based object ID (object ID 1, 2, 3) to the new content-based object ID (object ID K) by using an additional hashing as shown in Figure 6-(c). Thus, the distributed hash table will inform the location of given object associated with its content [21]. By employing this mechanism, we can remove the fingerprint index itself and preserve the scalability of the underlying storage system.

This mechanism gives following advantages. First, it gracefully removes the fingerprint index that can be a significant problem when applying deduplication to the scale-out distributed storage system. Second, it preserves the original scalability of underlying distributed storage system. Third, no modification is required on client side because the client request is based on the original object ID.

Self-contained object: We design a self-contained object for data deduplication to solve the external deduplication metadata problem. As described, the external design makes difficult for integration with storage feature support since existing storage features cannot recognize the additional external data structures. If we can design data deduplication system without any external component, the original storage features can be reused. Figure 7 shows the

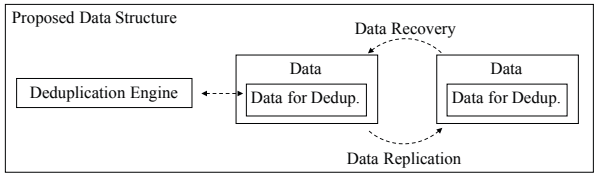


Figure 7: Deduplication data self-contained data structure.

concept of it. To this end, we extend the underlying storage's metadata to contain deduplication information. Note that, modern distributed storage systems have an extended attribute field (*xattr*) for this purpose.

Deduplication rate control & selective deduplication based on post-processing: Our key idea for minimizing performance degradation is post-processing with rate control and selective deduplication. In our design, background deduplication threads periodically conduct a deduplication job, and this background I/O is controlled through rate control. We also maintain the object's hotness, which can make sure that the hot object is not deduplicated until its state is changed.

As mentioned in Section 3.1, the inline deduplication has the advantage of space efficiency. However, it cannot avoid to exhibit an additional latency due to the inline processing. For example, in a distributed system, metadata lookup and data processing for deduplication chunk occurs in the network. Therefore, latency can be longer than the latency of metadata and data processing on a local node. In inline processing, this overhead can be huge because deduplication should be immediately executed. For this reason, inline processing is hard to guarantee the performance. On the other hand, we can achieve two major benefits by using post-processing. First, with ratio control, constant throughput is guaranteed. Post-processing can hide that latency problem because foreground I/O is handled as existing method and background deduplication thread executes a deduplication job later. However, the worst case should be considered that the foreground I/O job is interfered by the background deduplication tasks as shown in section 3-3. Therefore, if we add rate control technique, foreground I/O interference can be minimized. Second, it can give a chance that frequently modified object does not need to be deduplicated. In post-processing, background deduplication threads read the data and then, they conduct deduplication process. Therefore, we can control whether or not the hot object is deduplicated.

4. Design

4.1. Object

Storage features such as high availability, data recovery and various data management operations are per-object basis. Therefore, if we define all data deduplication information into an object, the underlying distributed storage system can handle the complicated storage features without an additional modification. In our design, an object is classified into two types based on the information stored in: metadata object and chunk object. Each of them has their own object metadata. Although chunk objects and metadata objects have a different purpose from existing objects, they can be handled the same way as existing objects because they are self-contained objects. Therefore, distributed storage system can handle distributed storage-dependent jobs such as replication, erasure coding or data rebalancing for each object without additional operations.

Metadata object: Metadata objects are stored in the metadata pool, which contains metadata for data deduplication. In a data deduplication system, data is divided into multiple chunks according to its chunking algorithm in detecting redundancy more effectively. The ID of the metadata object is a user-visible ordinary object ID, provided by the underlying distributed storage system. In a metadata object, a chunk map that links an object to chunks is stored based on its offset. As shown in Figure 8, a chunk map consists of offset range, chunk ID, cached bit and dirty bit. Offset range and chunk represent mapping information between metadata object and chunk object. Cached bit and dirty bit describe status of a chunk. If the cached bit is true, the chunk is stored in the metadata object. Otherwise, it is stored in the chunk object of chunk pool. A detailed explanation of the chunk pool is provided in Section 4.2. If the dirty bit is true, processing deduplication on the chunk is needed. If cached bits for all chunk map entry are false, there is no cached data in the object's data part. In figure 8, object 2 depicts this case. The type of object 2 is metadata and its chunk map represents that all of the chunks (object B and C) that consist the object are not cached. Thus, object 2 contains no data but only metadata. In contrast, if the cached bit is true, the chunk is stored inside of the object. In Figure 8, object 1 and 3 are this cases. Detailed policy on chunk caching is provided in section 4.3 cache manager.

Chunk object: Chunk objects are stored in chunk pool. Chunk object contains chunk data and its reference count information. In Figure 8, object B, C and D are chunk objects. A chunk data is stored in the data part of an object and reference count information (pool id, source object ID, offset) is stored in the metadata part of an object. The ID of a chunk object is determined by chunk's contents.

4.2. Pool-based Object Management

We define two pools based on the objects stored in. Metadata pool stores metadata objects and chunk pool stores chunk objects. Since these two pools are divided based on the purpose and usage, each pool can be managed more efficiently according to its different characteristics. Metadata pool and chunk pool can separately select redundancy scheme between replication and erasure coding depending its usage and each pool can be placed to different storage location depending on the required performance.

4.3. Cache Manager

Cache manager evaluates whether a chunk needs to be cached. If a chunk is cached, it is stored in the data part of the metadata object. By caching hot data, we can remove the deduplication overhead. However, in practice, caching an object from the chunk pool to the metadata pool needs a policy since storing an object causes extra I/O requests and storage capacity. For example, although there is a single

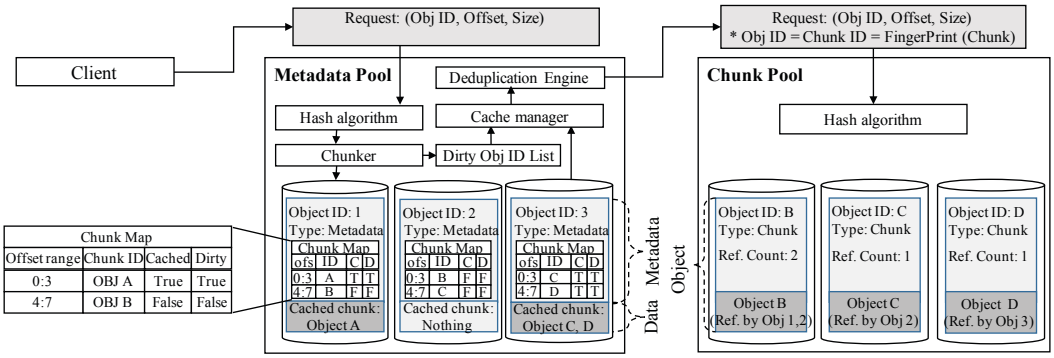


Figure 8: System design. The proposed method consists of metadata pool and chunk pool. Metadata pool contains metadata objects that contain deduplication metadata and cached chunks. Chunk pool contains deduplicated chunk objects. All object's location is determined by their own object ID. In case of chunk object, its object ID is generated by its contents using fingerprint hashing.

chunk in the chunk pool, metadata objects in the metadata pool can have duplicated chunks. Various cache algorithms [24] [30] could be applied here but in our experiment, we used a LRU based approach, which is simple.

4.4. Data Deduplication

As described in the key idea section, the proposed method mitigates the complexity of duplicate chunk detection in existing hash mechanisms. Specifically, a chunk is a basic unit for detecting redundancy of given data. When a data write request comes to a deduplication system, the data is split into several chunks. The chunk will be hashed, and the hashed value (i.e. chunk object ID) will be used as the input key for the hash algorithm of the underlying distributed storage system (i.e. chunk pool). As a result, if two chunks have the same contents, their location in the storage system is the same and it naturally removes the duplicates (*Double hashing*).

4.4.1. Deduplication engine. Since, we choose the post-processing deduplication, the engine is run by a background thread. The background deduplication engine begins deduplication with following steps.

(1) Find dirty metadata object which contains dirty chunks from the dirty object ID list. Note that, all modification or new write requests for an metadata object are logged into the dirty object ID list. Figure 8 shows the dirty object ID list.

(2) Find the dirty chunk ID from the dirty metadata object's chunk map. The dirty chunk is cached inside of the dirty object.

(3) If the cache manager judges that the dirty chunk is deduplication target, the deduplication engine checks whether the chunk entry corresponding to the dirty chunk already has a chunk object ID. If it has a chunk object ID, it is referenced by some chunk object earlier. Therefore, the deduplication engine sends old chunk object a de-reference message and wait for its completion. Then, a chunk object is generated and sent to the chunk pool. At the same time, chunk object ID is re-evaluated according to its contents. If the dirty chunk does not have a chunk object ID, the deduplication engine generates a chunk object and send it to the chunk pool.

(4) In the chunk pool, the chunk object generated in step 3 is placed in the underlying distributed storage system using the hash algorithm.

(5) If there is no object at the location which is determined by the hash algorithm, store the object with reference count = 1. If there is an object already stored at the location, add reference count information to the object.

(6) When the chunk write at the chunk pool ends, update the metadata object's chunk map. (Deduplication ends)

4.4.2. Deduplication rate control. The proposed post-processing deduplication system requires additional storage and network I/O (i.e. data transferring from metadata pool to chunk pool). So, background deduplication can directly affect on foreground I/O job. To minimize the interference by the background deduplication job, we control deduplication rate. Specifically, deduplication rate is controlled depends on pre-defined watermark value. In order to do that, we define low-watermark and high-watermark based on IOPS or throughput. For example, If IOPS is higher than high-watermark, a single deduplication I/O is issued per 500 foreground I/Os. And a single deduplication I/O is issued per 100 foreground I/Os between low-watermark and high-watermark. There is no I/O limitation if IOPS is lower than low-watermark.

4.5. I/O Path

Write path: Basically, write path is similar with the underlying distributed storage system because it is post-processing deduplication. Detailed write steps are as follows.

(1) Client issues an object write request with object ID, offset, size with data to the metadata pool.

(2) The hash algorithm in the distributed storage determines the location of the new object according the object ID and write the data. If the data size is less than the chunk size, the missing part is pre-read from the stored chunk object when cache bit is false and dirty bit is true.

(3) After the data is written in the data part of the object, a chunk map is created in the metadata part of the object. At the same time, chunk map entries are created and added to the chunk map. However, the chunk ID is not determined yet because it requires content based fingerprint hashing and

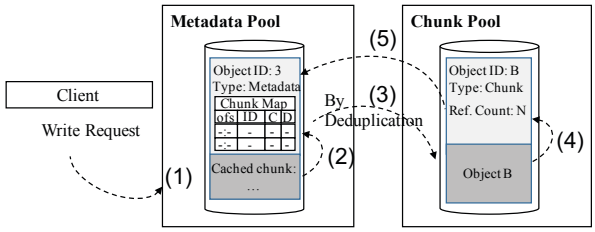


Figure 9: Object transaction based consistency support.

it takes an additional latency. The cached bit and dirty bit are set to true.
(4) Update the dirty object ID list for data deduplication. (Write path ends)

Read path: Read path is slightly different from the underlying distributed storage system since it needs to read metadata and its chunk from metadata pool and chunk pool, respectively. However, when a chunk is cached, the read path is similar with the original distributed storage system. Detailed explanation of read path is as follows.
(1) Client issues an object read request with object ID, offset and size to the metadata pool.
(2) The hash algorithm in the metadata pool determines the location of the requested object.
(3) Read the object's chunk map to find the exact chunk.
(4) a. If the chunk is cached, then read the chunk from the object's data part and return the chunk to the client. (Cached object read path ends)
(4) b. If the chunk is not cached, then read the chunk ID and issue a read request to the chunk pool using the chunk ID, an offset and a size. Finally, the object is returned to the client (Non-cached object read path ends)

4.6. Consistency Model

Since the proposed model separates metadata and data, consistency should be considered. In the proposed model, data consistency is achieved by the transactional operation of underlying storage system. Also, objects that reference chunks are tracked. In this manner, we can preserve the consistency correctly.

Figure 9 describes the consistency model step by step. A client requests a write operation (1). The data is written as the cached chunk and it's state is changed by dirty (2). These operations are a single transaction. Note that the chunk state is stored as the object's metadata. After that, deduplication process is triggered and dirty objects are found. These objects are flushed to chunk pool (3). Chunk pool stores flushed data and reference count information (pool id, source object ID, offset) as shown in (4). If the object is already existed, just reference count information will be added. Then, the deduplication result is sent (5), which in turn the state of dirty object is changed by clean. If failure occurs at (1), (2), the client can recognize the failure of write operation because it receives an ack or write timeout occurs. If failure occurs at (3), (4), chunk's state is not cleaned. Therefore, next deduplication process handles this dirty chunk. If failure occurs at (5), a chunk is stored in the chunk pool, but dirty state is not cleaned yet. In this case, the right result will also be sent when deduplication process is executed again. Since reference data is already stored in the chunk pool, if reference data already exists, the ack is sent without storing chunk and reference data.

The weak point of this model is the performance due to synchronous operations and the overhead of metadata size. For improving these weak points, we can use a technique named false positive reference count which strictly locks on increment but no locking on decrement [23]. However, this approach needs additional garbage collection process.

5. Implementation Notes on Ceph

We implemented the proposed deduplication method on top of Ceph.

Metadata pool and chunk pool support: Ceph supports tiering architecture and we can exploit the tiering to support metadata pool and chunk pool. Since most modern distributed storage systems support tiering architecture, it is easy to port to other storage systems.

Object metadata: We can simply add the additional metadata such as chunk map or reference counter using the xattr field and the key value store in the Ceph object metadata. Since modern distributed storage system supports xattr field or other similar fields, we can easily apply the proposed method into other storage systems. Each chunk entry in chunk map uses 150 bytes. Also, the object in chunk pool uses additional 64 bytes for reference and a map data structure for referenced objects. However, if chunk size is too small, such as 4KB, the number of reference objects and the size of chunk map will also increase. Therefore, the per-object space overhead also increases because Ceph's object has its own metadata at least 512 bytes.

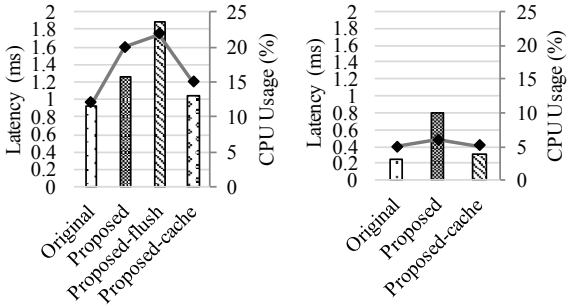
Chunking algorithm: We apply static chunking algorithm that uses a fixed size chunk because of its simple implementation and low CPU overhead. Note that small random write requires high CPU usage of around 60% to 80% on Ceph [40]. It is expected that if the CPU intensive algorithm such as Contents Defined Chunking is applied, the overall performance can be degraded because of CPU limitation [32].

Cache management (hitset & bloomfilter): Cache management is important for performance. We exploit the HitSet in Ceph for LRU based Hot/Cold cache. HitSet sustainably maintains recently accessed object set per second and counts for each object access. If an access count for an object is higher than pre-defined parameter *Hitcount*, then the object is cached into the metadata pool. Since the HitSet is stored in a storage, the in-memory bloomfilter is used for existence checking.

6. Evaluation

6.1. Environment setup

We implemented the proposed method on Ceph 12.0.2. For the experiment, a Ceph cluster is composed of four server nodes and each server has Intel Xeon E5-2690 2.6Ghz (12 cores) with 128GB of RAM and four SSDs (SK Hynix 480GB). Totally three client nodes are connected to the cluster with 10GbE NIC. Each server runs four OSD daemons configured with 2GB journal size and each OSD



(a) 8KB random write (b) 8KB random read

Figure 10: Performance of proposed post-processing deduplication (random write and random read). Bar indicates latency and solid line indicates CPU usage.

runs on the XFS local filesystem. For the replicated pool, replication factor is two. We configured static chunk size of deduplication as 32KB. Our evaluation is based on Ceph's block storage service, but the performance of object and file storage also would be similar because we implement our proposed design based on Ceph RADOS, a core I/O part that is common to all three storage systems.

6.2. Performance Comparison

6.2.1. Small Random Performance. Figure 10 shows the small random performance of the 32KB chunk size system. We measured the latency and CPU usage of random write and read in 8KB block size on a single client using FIO (4 threads, 4 iodepth). Original shows the results of the existing Ceph and *Proposed* shows the results of the proposed system with deduplication rate control. *Proposed*-flush shows the results when all data is written directly to the chunk pool, and *Proposed*-cache shows the results when data is stored in metadata pool first.

In the case of random write, *Proposed* 's latency increases up to 20% and CPU usage is doubled compared to the existing case. This is because additional works are required to complete I/O, such as writing metadata (i.e. chunk_map update), reading data for flush, generating fingerprints, and transferring data to the network. *Proposed*-flush's shows the worst result among all the results since the deduplication processing is executed immediately. *Proposed*-cache shows similar performance to Original. This is because its data exists in the metadata pool and only updates to the chunk_map occur.

In the case of random read, we compare the Original case, *Proposed* case (the data in the chunk pool), and *Proposed*-cache case (the data in the metadata pool). Since the redirection is needed (the client issues I/O request $\rightarrow$ the metadata pool forward client's I/O request to the chunk pool $\rightarrow$ the data is transferred from the chunk pool $\rightarrow$ the client receives an ack), *Proposed* 's latency is higher than the others. It can be seen that the random read of cached objects gives almost similar performance as Original.

6.2.2. Sequential Performance. Figure 11 is the result of testing FIO 32KB, 64KB, and 128KB sequential performance for 32k chunk size system. Throughput and latency were measured in three 10Gbit clients. All read tests were done after the data was flushed to the chunk pool.

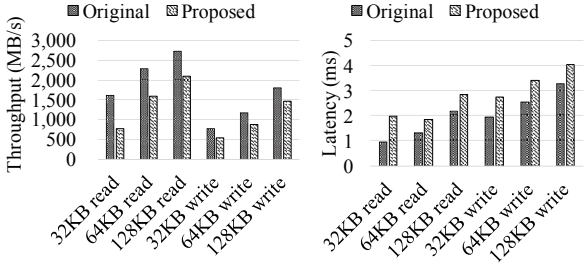


Figure 11: Sequential performance of proposed post-processing deduplication (read and write).

	16KB	32KB	64KB
Ideal dedup. ratio (%)	46.4	44.8	43.7
Stored data (TB)	1.82	1.88	1.89
Stored metadata (GB)	163	82	41
Actual dedup. ratio (%)	41.7	42.4	43.3

TABLE 2: Deduplication ratio comparison based on chunk size of 16KB, 32KB, and 64KB.

In the case of read, performance is reduced by half compared to Original when the block size is small. This is because the overhead of redirection (from the metadata pool to the chunk pool) increases in case of small block size (the less latency is required). However, when the block size is large, the overhead is relatively reduced. In the 128KB case, 32KB chunks are requested to the chunk pool in parallel, so that both throughput and latency are improved. Ideally, the read performance should be similar to the original. However, due to fragmentation caused by deduplication [26] [27] (sequential read becomes a random read), performance is degraded. Our design can not completely remove this overhead. However, with cache manager, serious performance degradation will be prevented because hot object is handled in the metadata pool.

The write performance is measured based on the high-watermark value. Since the deduplication is performed at a constant rate in the metadata pool, there is only limited performance degradation compared to the original performance regardless of the block size requested by the client.

6.3. Space Saving

Table 2 shows the results of deduplication ratio based on the chunk size. The experiment is performed on our private cloud shown in Section 2 and all the results are calculated under excluding the redundancy caused by replication. Ideal deduplication ratio means deduplication ratio of data only. It can be observed that deduplication ratio gets lower as the chunk size gets bigger. Actual deduplication ratio is calculated by including size of metadata appended to data. In the proposed method, metadata for metadata objects and chunk objects is additionally required as explained in Section 4. The size of this additional metadata grows proportionally as the chunk size becomes smaller. So, although the smallest chunk size shows the highest data deduplication ratio, its actual deduplication ratio is the lowest. On the other hand,

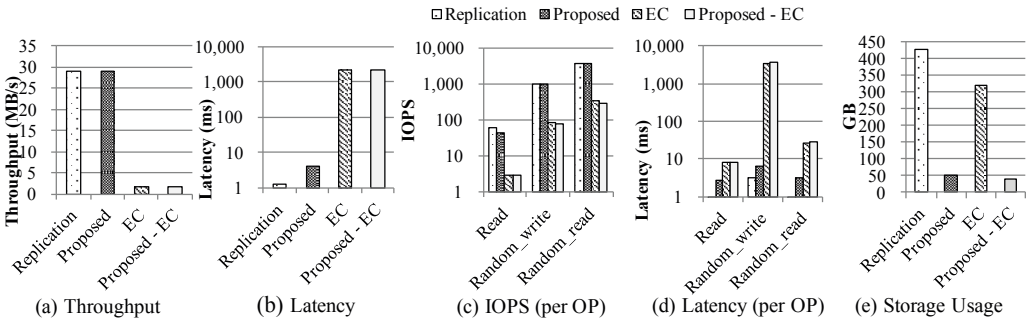


Figure 12: SPEC SFS 2014 database workload evaluation. Proposed means proposed method with replication. Proposed-EC means proposed method with EC. Y-axes for Latency and IOPS are logarithmic scale.

if the chunk size is large, the data deduplication ratio is reduced, which reduces the overall deduplication ratio again.

6.4. Deduplication Synergy Effect with Storage Features

6.4.1. High Availability with Deduplication. In this evaluation, we show the detailed results on high availability using database workload from SPEC SFS 2014 benchmark. A client created a block device through the KRBD module and then evaluated with this block device. For experiment, we set replication scheme configured with replication factor of two and erasure coding (EC) scheme configured with k=2, m=1. The workload metric was set to 10, resulting in a total size of 240GB files. Figure 12 shows various evaluation results using SPEC SFS 2014. Note that the workload of SPEC SFS is mixed with read, random read, and random write simultaneously.

Performance: In Figure 12-(a), the total throughput is similar in replication and the proposed method. However, the throughput of EC and *Proposed*-EC are significantly lower than those. Note that the database workload in SPEC SFS 2014 issues fixed number of requests per second. That's why there is no difference between replication and the proposed method. Figure12-(b) shows the total latency result. Replication shows about 1.26 ms, while the *Proposed* method shows to 4.1 ms due to the deduplication processing overhead. In contrast, EC and *Proposed*-EC show latency of 2s.

Figure 12-(c) depicts IOPS result for the evaluation of each operation. Similar to other results, the EC and the *Proposed*-EC show low IOPS.

The latency for each operation can be seen in Figure 12-(d). In the case of EC random write, parity calculation is required unlike replication, and read-modify-write according to write size is required, which is slower than the result of the *Proposed* method. In the case of read, it can be seen that EC is relatively small in latency gap with the proposed method due to influence of read-ahead and cache. However, in case of random read, it is difficult to see the effect of read-ahead or cache so that higher latency occurs due to simultaneous reading of widely spread chunks among several nodes. Therefore, the latency gap between the

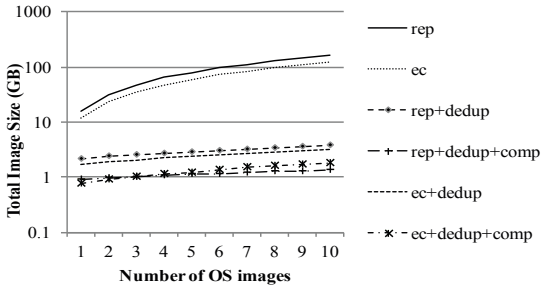


Figure 13: Combination with data compression. The proposed method can combine with data compression features of underlying storage system. Totally ten 8GB of Ubuntu Linux VM images are used for the measurement.

Proposed and *Proposed*-EC was greater than in the case of reading.

Storage Saving: Figure 12-(e) shows space usage under evaluation using SPEC SFS 2014. The replication method uses 428GB total, but for EC, 320GB was used. The *Proposed* method used only 48GB. In SFS workload that represents the real workload, the significant storage saving by data deduplication can be observed.

	Failed OSDs (#)		
	1	2	4
Original	68.04	71.35	81.77
Proposed	43.72	44.51	54.78

TABLE 3: 100GB data recovery time measurements (in seconds) with respect to the increased number of failed OSDs.

6.4.2. Data Recovery Acceleration with Deduplication. Table 3 shows the recovery time while removing and re-adding the OSD on each node. The failed OSD indicates the number of OSDs that have been removed and added in order to simulate OSDs failure. Totally 100GB of data was stored at 50% deduplication ratio using replication and the recovery time was compared between the *Proposed* case and Original case. In the case of applying *Proposed* , data size to be recovered is 50% smaller than Original case due to data deduplication. As a result, *Proposed* can make significant benefit for the data recovery time.

6.4.3. Combination with Data Compression For Maximized Capacity Saving. One of the common ways to save

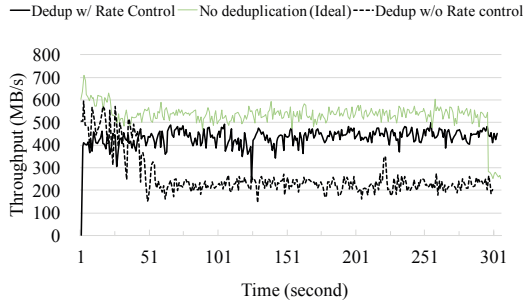


Figure 14: Dedup rate control

storage space is data compression. Modern file systems such as ZFS or Btrfs natively support this feature. Since a Ceph node works on a local file system, a Ceph node can use data compression feature. We can maximize storage saving with data compression feature. For this experiment, we use Btrfs for local filesystem. Figure 13 shows the combined result of capacity reduction between replication, EC, compression and deduplication. Totally ten 8GB of Ubuntu VM images that are running in our private openstack cinder service. The OS images are the same but user home data are different. The x-axis represents the cumulative number of OS images and the y-axis represents the total occupied space. Note that the y-axis is logarithmic scale. Replication occupies 160GB of space (8GB x 10 images x 2 replications). For EC, we set 2+1 configuration that uses 2 disks for data and 1 disk for reconstruction data. For EC 2+1, therefore, totally 120 GB are occupied. In the experiment, the actual footprint when using the proposed data deduplication with replication is approximately 2.2GB. In addition, when one VM image is added to the cluster, only about 200MB is added. This is because there is lots of redundant data between VMs using the same OS. As depicted, EC + deduplication + compression achieves maximum capacity saving.

6.5. Deduplication Rate Control

We could minimize the foreground I/O job interference by background deduplication processing using deduplication rate control (pre-defined high-watermark value is used). Figure 14 shows throughput of a foreground thread that issues sequential write while background deduplication job is processing. When there is no deduplication (Ideal case-green solid line) throughput goes around 500-600MB/s. However, it significantly decreased to 200MB/s when a background deduplication job is processing (black dotted line). The proposed deduplication rate control shows almost 400-500MB/s even if there is a background deduplication job (black solid line).

7. Related Works

HYDRAstor [35], HydraFS [36] designed a distributed storage system that can deduplicate via content-addressed and DHT similar to this paper. However, the following points are different: First, in those papers, content-based hash must be created in the client library or filesystem for addressing. On the contrary, this study not only generates hash by post-processing but also maintains the I/O path used in existing shared-nothing storage system. Therefore, the client does not cause latency due to hashing and can maintain the performance of underlying system by selectively performing deduplication. Second, although [35] [36] designed its own data placement and recovery function based on DHT, in this paper, it is possible to make dedup data based on existing object, so recovery and rebalance function can be re-used. Therefore, a reliable service can be easily applied. Third, the target of this paper is the existing shared-nothing scale-out storage system which does not consider deduplication. Fourth, [35] [36] mainly consider throughput and are based on backup system. On the other hand, this paper focuses on not only throughput but also latency for primary storage.

Venti [21], Sean C. Rhea et al. [22] used the same method in that it uses the fingerprint value generated from data content as the OID. However, the following points are different. First, because [21] manages metadata and data separately, a mechanism for recovery and re-balance is required. On the other hand, this paper treats both metadata and data as objects, so it utilizes the recovery function of existing storage. Second, [21] should generate the fingerprint of the data content unconditionally in order to specify the address, but this paper can gain the performance advantage because it can store the data in the metadata pool and decide to perform the deduplication selectively. Third, [21] is an archival storage and it is used only for specific purposes with low performance. However, in this paper, it can be applied to existing block, file and object interface structure by optimizing performance and ensuring consistency. Fourth, existing CAS-based storage systems, including [21] [22] are centralized and therefore sensitive to the performance of the central server. However, in this paper, the scale-out performance is guaranteed.

Austin T. Clements et al. [1] implemented deduplication between VMFSs. They also propose the design without metadata server. However, only the VMFS environment is considered, not the distributed storage system, and the data location is defined by defining a separate block mapping for the VM image.

8. Conclusion

In this paper, we present a global deduplication design for current shared-nothing scale-out storage system, which can be combined with existing storage features such as high availability, data recovery while minimizing performance degradation.

9. Acknowledgement

This research was partly supported by National Research Foundation of Korea (2015M3C4A7065646).

References

[1] Austin T. Clements, Irfan Ahmad, Murali Vilayannur, Jinyuan Li: Decentralized Deduplication in SAN Cluster File Systems. USENIX Annual Technical Conference 2009

[2] K. V. Rashmi, Mosharaf Chowdhury, Jack Kosaian, Ion Stoica, Kannan Ramchandran: EC-Cache: Load-Balanced, Low-Latency Cluster Caching with Online Erasure Coding. OSDI 2016: 401-417

[3] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin. Erasure coding in Windows Azure Storage. In USENIX ATC, 2012

[4] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kuamr. f4: Facebook's warm BLOB storage system. In OSDI, 2014.

[5] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Yucheng Zhang, Yujuan Tan: Design Tradeoffs for Data Deduplication Performance in Backup Workloads. FAST 2015: 331-344

[6] Biplob K. Debnath, Sudipta Sengupta, Jin Li: ChunkStash: Speeding Up Inline Storage Deduplication Using Flash Memory. USENIX Annual Technical Conference 2010

[7] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, Sudipta Sengupta: Primary Data Deduplication - Large Scale Study and System Design. USENIX Annual Technical Conference 2012: 285-296

[8] Dutch T. Meyer, William J. Bolosky: A Study of Practical Deduplication. FAST 2011: 1-13

[9] Joo Paulo, Jos Pereira: Efficient Deduplication in a Distributed Primary Storage Infrastructure. TOS 12(4): 20:1-20:35 (2016)

[10] Tianming Yang, Hong Jiang, Dan Feng, Zhongying Niu, Ke Zhou, Yaping Wan: DEBAR: A scalable high-performance de-duplication storage system for backup and archiving. IPDPS 2010: 1-12

[11] Xun Zhao, Yang Zhang, Yongwei Wu, Kang Chen, Jinlei Jiang, Keqin Li: Liquid: A Scalable Deduplication File System for Virtual Machine Images. IEEE Trans. Parallel Distrib. Syst. 25(5): 1257-1266 (2014)

[12] Wen Xia, Hong Jiang, Dan Feng, Yu Hua: SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput. USENIX Annual Technical Conference 2011

[13] Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google file system. SOSP 2003: 29-43

[14] Lustre, http://lustre.org/

[15] pNFS, http://www.pnfs.com/

[16] Glusterfs, https://www.gluster.org/

[17] Swift, https://docs.openstack.org/swift/latest/

[18] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, Carlos Maltzahn: Ceph: A Scalable, High-Performance Distributed File System. OSDI 2006: 307-320

[19] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, Werner Vogels: Dynamo: amazon's highly available key-value store. SOSP 2007: 205-220

[20] Scality, http://www.scality.com/

[21] Sean Quinlan, Sean Dorward: Venti: A New Approach to Archival Storage. 89-101

[22] Sean C. Rhea, Russ Cox, Alex Pesterev: Fast, Inexpensive Content-Addressed Storage in Foundation. USENIX Annual Technical Conference 2008: 143-156

[23] Zhuan Chen, Kai Shen: OrderMergeDedup: Efficient, Failure-Consistent Deduplication on Flash. FAST 2016: 291-299

[24] Wenji Li, Gregory Jean-Baptise, Juan Riveros, Giri Narasimhan, Tony Zhang, Ming Zhao: CacheDedup: In-line Deduplication for Flash Caching. FAST 2016: 301-314

[25] Kiran Srinivasan, Timothy Bisson, Garth R. Goodson, Kaladhar Voruganti: iDedup: latency-aware, inline data deduplication for primary storage. FAST 2012: 24

[26] Benjamin Zhu, Kai Li, R. Hugo Patterson: Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. FAST 2008: 269-282

[27] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat: Improving restore speed for backup systems that use inline chunk-based deduplication. FAST 2013: 183-198

[28] Bo Mao, Hong Jiang, Suzhen Wu, Lei Tian: POD: Performance Oriented I/O Deduplication for Primary Storage Systems in the Cloud. IPDPS 2014: 767-776

[29] Permabit, http://permabit.com/

[30] Jingwei Ma, Rebecca J. Stones, Yuxiang Ma, Jingui Wang, Junjie Ren, Gang Wang, Xiaoguang Liu: Lazy exact deduplication. MSST 2016: 1-10

[31] Zhen Sun, Geoff Kuenning, Sonam Mandal, Philip Shilane, Vasily Tarasov, Nong Xiao, Erez Zadok: A long-term user-centric analysis of deduplication patterns. MSST 2016: 1-7

[32] Yinjin Fu, Hong Jiang, Nong Xiao: A Scalable Inline Cluster Deduplication Framework for Big Data Protection. Middleware 2012: 354-373

[33] Mark Lillibridge, Kave Eshghi, Deepavali Bhagwat, Vinay Deolalikar, Greg Trezis, Peter Camble: Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. FAST 2009: 111-123

[34] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel. A Study on Data Deduplication in HPC Storage Systems. In SC'12, Nov. 2012.

[35] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, Michal Welnicki: HYDRAstor: A Scalable Secondary Storage. FAST 2009: 197-210

[36] Cristian Ungureanu, Benjamin Atkin, Akshat Aranya, Salil Gokhale, Stephen Rago, Grzegorz Calkowski, Cezary Dubnicki, Aniruddha Bohra: HydraFS: A High-Throughput File System for the HYDRAstor Content-Addressable Storage System. FAST 2010: 225-238

[37] Deepavali Bhagwat, Kave Eshghi, Darrell D. E. Long, Mark Lillibridge: Extreme Binning: Scalable, parallel deduplication for chunk-based file backup. MASCOTS 2009: 1-9

[38] FIO, https://github.com/axboe/fio

[39] SPEC SFS 2014, https://www.spec.org/sfs2014/

[40] Myoungwon Oh, Jugwan Eom, Jungyeon Yoon, Jae Yeun Yun, Seungmin Kim, Heon Y. Yeom: Performance Optimization for All Flash Scale-Out Storage. CLUSTER 2016: 316-325