Towards Cluster-wide Deduplication Based on Ceph

Jinpeng Wang[†‡], Yang Wang[†], Hekang Wang[†‡],
Kejiang Ye[†], Chengzhong Xu[†♮], Shuibing He[♭], Lingfang Zeng[♯]
[†]*Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China*
[‡]*University of Science and Technology of China*
[♮]*University of Macau, China*
[♭]*Zhejiang University, China*
[♯]*Huazhong University of Science and Technology, China*
yang.wang1@siat.ac.cn

***Abstract*—In this paper, we design an efficient deduplication algorithm based on the distributed storage architecture of *Ceph*. The algorithm uses on-line block-level data deduplication technology to complete data slicing, which neither affects the data storage process in *Ceph* nor alter other interfaces and functions in *Ceph*. Without relying on any central node, the algorithm maintains the characteristics of *Ceph* by designing a special *hash object* to store the data fingerprint, and uses the CRUSH algorithm to judge the data duplication based on calculation, instead of global search. The algorithm replaces the duplicate data with the deduplicated objects, which storage their fingerprints with less storage space. We compare the effects of different block sizes with respect to the performance and deduplication rates through experimental studies, and select the most appropriate block size in our prototype implementation. The experimental results show that the algorithm can not only effectively save the storage space but also improve the bandwidth utilization when reading and writing the duplicate data.**

***Keywords*-deduplication; distributed storage system; Ceph**

I. Introduction

Ceph [1] is an open source distributed storage system, implementing object storage on a distributed computer cluster, and providing interfaces for object-level, block-level and file-level storage. By virtue of its stable operation mode, *Ceph* is being used by more and more enterprise users, and its decentralized control makes great contribute to its excellent scalability. With the expansion of *Ceph* cluster as well as the growth up of diverse data sets, there, as expected, would be more and more duplicate data in the storage cluster. Unfortunately, *Ceph* does not provide a solution to duplicate data managements [2]. In order to solve this problem, deduplication technology has been proposed [3]. However, traditional deduplication technologies often exploit the features of stored data to maximize the deduplication effects [4]–[6], ignoring the characteristics of the underlying storage platform. For example, Chen et al. [4] introduced a deduplication approach to duplicate images based on *Haar* wavelet while Zeng et al. [5] propose an online framework for VM image backup and recovery. In contrast to these two studies focusing mostly on backup data, Meister et al. [6] analyzed over one PB (1212 TB) of online file system data, and presented the study on the potential of data deduplication in HPC centers. All these methods, though effective toward its own targeted data, lack the consideration of the underlying platform features. To effectively solve the problem of data duplication in the cluster, we argue that the deduplication technique should not only leverage the stored data nature but also take full advantage of the platform features as the deduplication is a holistic system and its effective implementation for the data availability is highly platform-dependent. As such, given the complex data processing-flow in *Ceph*, say its multi-layer data mapping relationships, how to ensure the integrity of the data and achieve efficient data accesses is one of the difficulties in deduplication design for *Ceph* [7]. On the other hand, *Ceph* guarantees the security of stored data by storing multiple copies of data or using erasure code that require extra storage space. Hence, the balance between the data security and deduplication is also worth considering.

With these challenges in mind, we adopt FUSE technology [8], a most widely used user-space file system framework, to design and implement deduplication algorithm for *Ceph* in this paper. Although FUSE framework is rather controversial in file system implementation due to its performance overhead, it is still worthwhile to use this technique to quickly prototype our deduplication technique as a proof of concept. First, *CephFS* can be mount as a FUSE in user space, ensuring that the client host has a copy of the *Ceph* configuration file. As such, compared with in-kernel approach, using FUSE dramatically simplifies our development effort. Second, as more new features (e.g., zero-copying, writeback caching) are introduced to *Ceph*, its performance has been substantially improved and even comparable to that of in-kernel file systems when large, sustained I/O is performed [9], [10], which entails FUSE to be an adequate solution to the deduplication implementation for *Ceph*.

In summary, we make the following contributions:

1) We design a deduplication algorithm for *Ceph*, which fits to the architecture of *Ceph* distributed storage

system. The algorithm can complete the deduplication in the process of storing data in *Ceph* without affecting the object storage and block storage interface of *Ceph*.

2) We propose a concept of hash object, which is based on the *Ceph* data object. By using the data structure of the hash object, the algorithm could store the fingerprint of data distributed in the cluster nodes without relying on any central node.
3) By leveraging the core *CRUSH* (*Controlled Replication Under Scalable Hashing*) algorithm in *Ceph*, our deduplication algorithm could judge duplicated data by calculation instead of searching.

The remainder of this paper is organized as follows. We describe the background knowledge regarding *Ceph* storage in Section II. We propose the deduplication algorithm based on *Ceph* in Section III, and present the experimental results in Section IV, followed by reviewing some related work in Section V for comparison study. Finally, we remark and conclude the paper in the last section.

II. *Ceph* DISTRIBUTED STORAGE

Ceph is an open source distributed storage system, built on top of commodity components, demanding reliability to the software layer. A *Ceph* storage cluster consists of at least a *MON* (*Monitor*), *MDS* (*Metadata Server*),[1] and a *OSD* (*Object Storage Daemon*) as shown in Fig. 1 [1]. The *MON* managements the lifecycle of *OSD*s and is responsible for managing the authentication between the daemons and the clients. The *MDS* stores the metadata on behalf of the *Ceph* file system while the *OSD* stores the data, handles the data replication, recovery, rebalancing, and provides some monitoring information as well to the *MON* by checking other *OSD*s based on heartbeats.

Ceph offers three ways to use: *Ceph* file system (*CephFS*), *Ceph block device* and *Ceph object gateway*. The *CephFS* is a POSIX-compliant file system that uses a *Ceph storage cluster* to store its data. A block is a sequence of bytes block. Device interfaces makes a virtual block device an ideal candidate to interact with a mass data storage system like *Ceph*. *Ceph object gateway* is an object storage interface providing applications with a RESTful gateway to *Ceph storage clusters*.

The *Ceph* storage file process is shown in Figure 1. When *Ceph* stores a file, the client requests a unique file number *ino* (inode) from the *MDS* at first, and then divides the file into a sequence of fixed-size chunks, according to the object size set by user. After that, the client encapsulates the divided data into *data objects* managed by *Ceph*. The object name is composed of *ino* and *ono* (object number). Then the object is mapped to *PG* (*Placement Group*) through a hash calculation, here the *PG* is a logical layer between the object and the *OSD* map. Finally, the client calculates which *OSD* the object should be stored through the *CRUSH* algorithm [11], and store the object into the mapped *OSD* [12]. This write process is repeated for each object until all objects of the file are stored.

[1]*MDS* is only needed by *CephFS*.

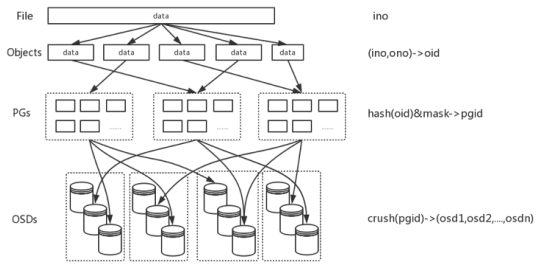

Figure 1: *Ceph*FS file storage process

By following this description, one can derive how read and other file operations are processed. More details can be found in [1].

III. DEDUPLICATION ALGORITHM DESIGN

A. Overview

Our deduplication algorithm is designed to solve the data duplication problem in the *CephFS* with consideration of its features. As such it needs to fit the *Ceph* storage architecture and have no impact on the interface of *Ceph block device* and *Ceph object gateway*. *Ceph* nodes are equal without central control, and the data fingerprint storage is one of the focuses in the algorithm. As stated, when it stores files, *Ceph* would divide the files into a sequence of fixed-size data objects and store them in the cluster. Therefore, this storage process is very friendly to the fixed-size chunking scheme in traditional deduplication technology. We borrow the idea from the data block level deduplication algorithm in our design, which centers around a concept of *hash object*. The structure of this object helps the algorithm accomplish the fingerprint storage and redundancy detection.

B. Data Object Structure Design

There are three basic functions to be implemented: 1) recording the fingerprint of stored data; 2) recording the number of times the data is accessed; 3) restoring the deduplicated data. In order to accomplish these functions, the algorithm designs three types of objects that modifies name and content and adds extended attributes of the *Ceph* object. The object design is shown in Figure 2.

1) *Ceph object:* we add some extended attributes based on the original object structure in *Ceph*: *flag*, *reference count*. The flag is used to identify the object as the original data object while the reference count is used to record the number of accesses to the object, with an initial value of zero.

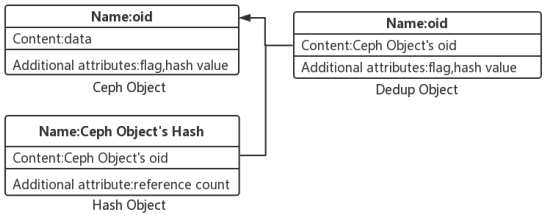

Figure 2: Data object structure design

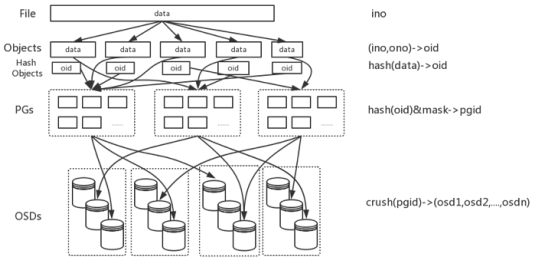

Figure 3: File stored process with deduplication

2) *Hash object:* The hash object is the key to the deduplication algorithm, which is generated for each *Ceph* object and used to record the hash value of the *Ceph* object content. The hash object name is the hash value of the content of the *Ceph* object, and its content is the object name of the *Ceph* object. Since the CRUSH algorithm can calculate the disk location of the object, we use it to detect the redundant data. While writing a new object, the hash value of the object content is calculated. We only need to perform the CRUSH algorithm on the hash value, and find if there is an object named in the hash value exists in the obtained disk location to finish the redundancy detection.
3) *Deduplication object:* if it is found that the current object's hash value already exists, then the current object stores the data as redundant data. We thus replace the content of the current *Ceph* object with the original object's oid that is duplicated to the object content, and add the duplicate data object with an extended attribute, flag, which is used to identify the current object as a deduplication object. The deduplication object and its hash object are only $20B$, which can save a large storage space.

C. Storing procedure

The storing procedure of the deduplication algorithm designed in this paper is shown in Figure 3. The storing procedure proposed in this paper is similar to the original one in *Ceph*. When obtaining the file to be stored, the client requests a unique *ino* for the file from the *MDS*. Then, the client divides the file into a sequence of fixed size data block, according to the object size set by user. After that, the client calculates the hash value of the current data block, and then detects whether the current data block stores duplicate data based on the hash value. The client creates a deduplicated object or a *Ceph* object, according to the detection result. The objects we created are compatible with the original data structure of *Ceph* object. *Ceph* will store the deduplicated object and its hash object as common *Ceph* objects in the subsequent processing flow.

D. File operations

This section introduces the specific process of the deduplication algorithm with respect to file write, read and delete. Note that all these operations are mainly designed for read-only data, a commonplace in practice, and performed at the *CephFS* client side. Update in place is a very complicated operation in data deduplication, which is left as our future study.

1) Writing files: The process of writing files is shown in Figure 4, where four steps are followed:

- After reading the file information, the client divides the file into a sequence of fixed size data blocks, according to the file size and the pre-set block size, except for the last data block, whose size is indeterministic. Then the client encapsulates the data blocks into the *Ceph* manageable data objects, called *Ceph object*.
- The client uses the SHA-1 algorithm to calculate the hash value of each *Ceph object* content, then creates an object, called *hash object* for each *Ceph object*. The hash value is used as the hash object name, and the oid of the original *Ceph object* is used as the hash object content. The hash objects are also managed by *Ceph*.
- For each *hash object*, the client calculates its *OSD* number by performing the CRUSH algorithm based on the hash object name and the cluster state. The calculated *OSD* is used to store the hash object. Then the client tries to find if there is an object with the same name. If not found, the content stored in the current *Ceph object* is unique, then the following operations will be performed:
  - add an extended attribute "reference count" for the hash object to record the access number of the *Ceph object*, whose initial value is set to 1;
  - add additional attributes "flag" and "hash value" for the *Ceph object*, used to mark the current object as a *Ceph object* and record the hash value of the object content.

  Otherwise, if there is an object that has the same name, the the content stored in the current *Ceph object* is redundant, then the following operations will be performed:

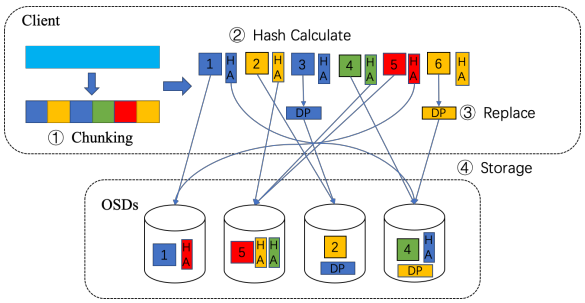

Figure 4: File write process

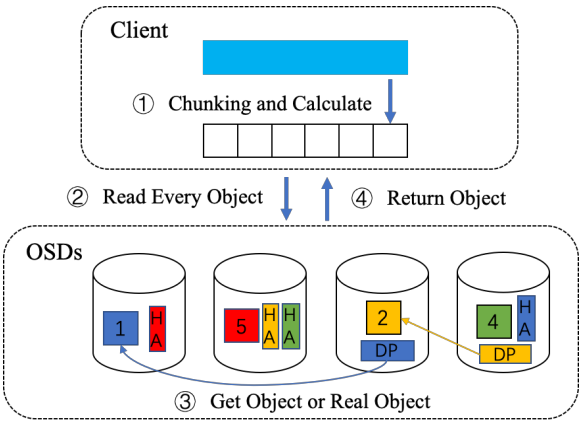

Figure 5: File read process

  - add a write lock to the hash object in the cluster, increase the value of the reference count by 1 and then release the write lock;
  - replace the contents of the current *Ceph object* with the content hash value, this *Ceph object* is called a *deduplicated object*;
  - add additional attributes "flag" and "hash value" to the *deduplicated object*, which is used to mark the current object as a *deduplication object*, and record the hash value of the object content.
- For the object generated in Step3, calculating the OSD number that the object should be stored based on the object name, the cluster information, and the CRUSH algorithm. Then the client directly communicates with the corresponding *OSD*, and stores the object into the *OSD* until all the files are divided and the divided objects are stored.

2) Reading files: The process of reading files is shown in Figure 5 where the specific processing steps are as follows:

- After receiving the request for reading an file, the client obtains the file's *ino* from the metadata server *MDS* according to the storage location of the file, and then calculate the *oid* of all the objects that store all the file contents.
- For each object the client wants to read, the client directly establishes a connection with the corresponding *OSD*.
- The *OSD* reads the extended attribute of the object. If the extended attribute identifies that the current object is the original object, the content of the object is read directly and return to the client. Otherwise, the current object is a duplicate object, and the content stored in the duplicate object is read, which is the *oid* of the original object. Then the content of the original object is read and return to the client, according to its *oid*.
- After reading all the object contents, the client concatenates the contents together into a file and returns it to the user.

3) Deleting files: As with the write and read operations, the process of deleting files shown in Figure 6 is described as follows:

- When receiving a request to delete a file, the client obtains the *ino* of the file from the *MDS*, according to the user request, and calculates the *oids* of all the file objects.
- The client calculates the numbers of *OSD*s where the file objects are stored, and then sends a deletion request to those *OSD*s, finally deletes the metadata of the file.
- After the *MDS* deletes the metadata of the file and the *OSD* receives the deletion request, the *OSD* marks all the objects to be deleted, and returns the deletion success response to the client. The *OSD* will accomplish the object deletion in due course.
- The *OSD* deletes the marked objects in due course. Different operations are performed on the objects of different types and states. Four operations determined by the extended attribute of objects are shown in Figure 7:
  1) The object is the original data object and its *reference count* is 0: Find the location where the hash object is stored, according to the hash value of the current object, and delete the data object after the deletion of the *hash object*;
  2) The object is the original data object and the *reference count* is not 0: Add a write lock to the *hash object*, and decrements the *reference count* by 1 and then release the write lock.
  3) The object is a duplicated object, and the corresponding original data object has a *reference count* of 0: First, find and delete its *hash object*, according to the hash value stored in the duplicated object. Then, find and delete the *Ceph object* by its *oid*, which is the contents stored in duplicated object, and delete the duplicated object eventually.
  4) The object is a duplicated object and the original data object *reference count* is not 0: First, find the *hash object*, according to the hash value stored in

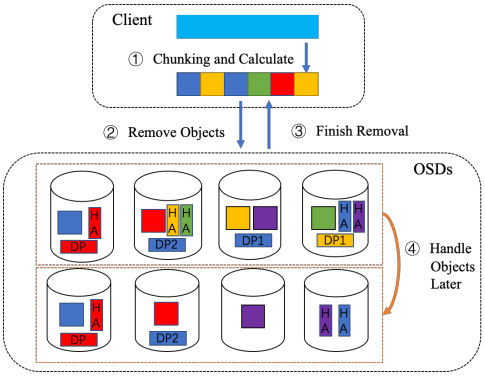

Figure 6: File read process

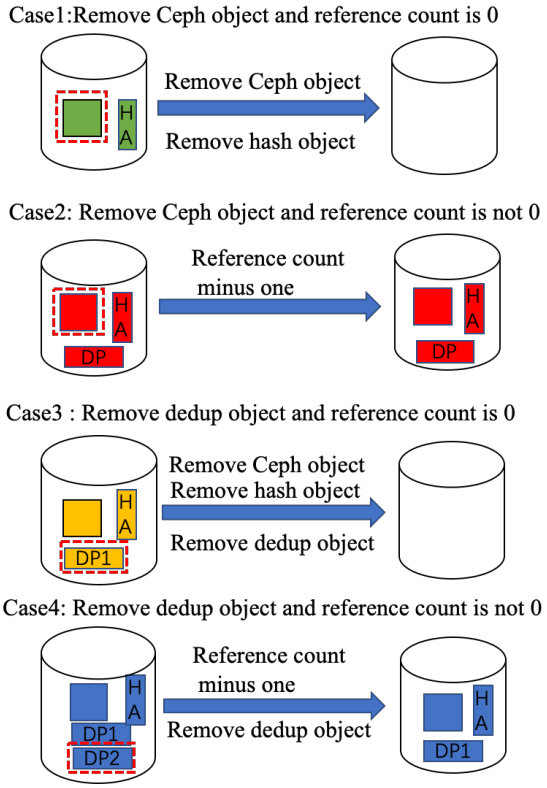

Figure 7: Object deletion with four different states.

the duplicated object, then add a write lock to the *hash object*, and release the write lock after decrementing the *reference count* by 1. Finally, delete the deduplicated object.

IV. Performance Evaluation

A. Experimental environment

The network topology of the experimental environment is shown in Figure 8. All nodes are with the same configuration as shown in Table I. The *Ceph* storage cluster is built using three nodes. In terms of role assignment, three nodes are used as the *MON*, two nodes as the *MDS*, and four hard

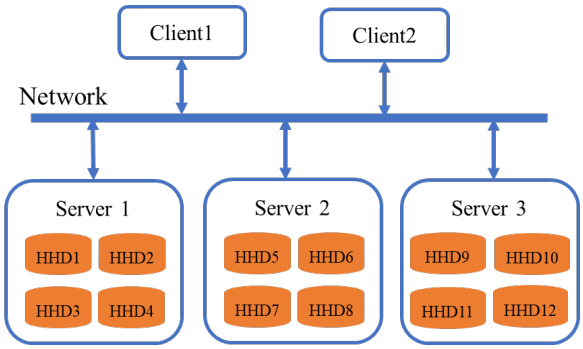

Figure 8: Test environment network topolog.

Table I: Cluster node configuration

Testbed	Configuration
CPU	E5-2630 v3
Memory	64GB
Network Bandwidth	10Gbps
Hard Disk	Seagate ES.3: 4TB *4
Operating System	Centos 7.2
Linux Kernel Version	4.4.13
Ceph Version	Jewel 10.2.0
Fuse Version	2.9.2

disks of three nodes as the *OSD*s. The other two nodes are used as client nodes, and the network between the client nodes and the *Ceph* cluster is 10 Gigabit network.

B. Experimental method

The client mounts the *Ceph* cluster to the local file system through FUSE, and conducts the performance tests of file reading and writing through the FIO test tool [13]. The storage cluster adopts a dual-copy mode To ensure the data security where two copies of the file are reserved each time the file stored. We first use the *Ceph* native client to perform file read and write tests, and then exploit the modified *Ceph* client with the deduplication function for comparison.

The file in the experiments is a selected Linux image file: `CentOS-7-x86_64-DVD-1511.iso`. When enabling the original *Ceph* to write or read a file, the tests are made in the experiments by using FIO based different IO block sizes. In comparison, when writing or reading a file by the deduplication client, the test file is repeatedly written three times in different block sizes via FIO. The following experiments demonstrate the bandwidth when reading and writing files and the CPU utilization of the client nodes.

C. Experimental results and analysis

1) Storage usage: How the storage is used to access the test file is shown in Figure 9 where the first bar in each group represents the total storage in the cluster, including the file data, replicas and the metadata, that is occupied by the default client when different I/O data block sizes are

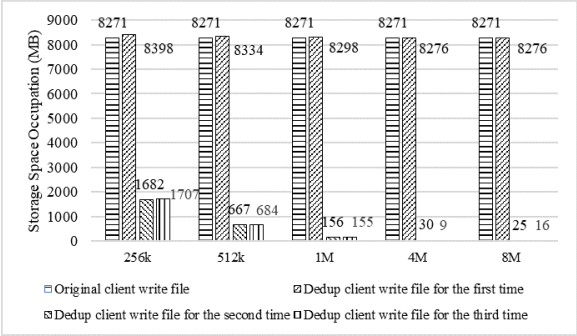

Figure 9: Storage usage while writing the same file multiple times with different block sizes.

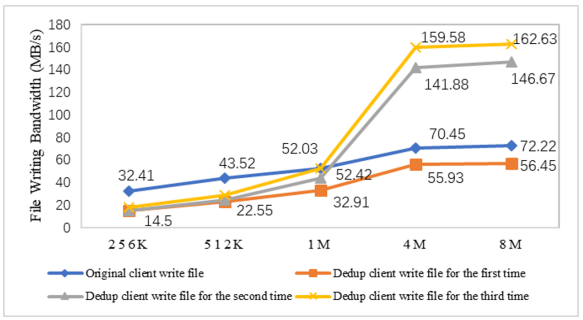

Figure 10: Network bandwidth while writing the same file multiple times with different block sizes.

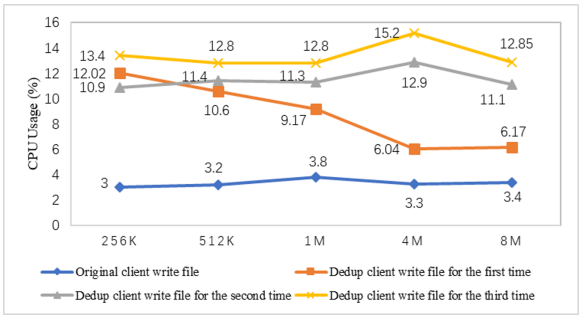

Figure 11: CPU utilization while writing the same file multiple times with different block sizes.

configured, while the other three bars in each group are from the deduplication clients using different block sizes for comparison. Specifically, the second bar represents the storage used by the deduplicate client in the first time. The results indicate the redundant data are quite rare and the overhead incurred by the deduplication (say the hash object) is marginal since the total used storage is only increased slightly. The third and fourth bars represents two consecutive writes of the file after the first time, in which the deduplicate client deletes the redundant data so that the file takes up less storage space. As we can see from the figure, the smaller the block size, the larger the total cluster space occupied. These results are easy to understand as the smaller objects are divided, the more *hash objects* are generated, which will take up more extra space. In particular, when the block size is 256KB, it can on average save 79.7% of the cluster storage space, while when the block size is increased to 4MB, it can save up to 99.6% of the cluster storage space.

2) Bandwidth usage: The bandwidth results of the written file is shown in Figure 10. The blue line is the bandwidth change when the original client writes the file by defaults with different I/O data block sizes while the other three lines, similar to in storage experiment, are the results of the deduplication client, which runs three times, each with a different block size. It can be seen from the figure that the bandwidth differences between the blue line and the brown line are approximately a small constant, which demonstrates the network overhead due to the deduplication is small. Moreover, the yellow and grey lines represent the duplication client writes, and their bandwidths are substantially increased when the block sizes grow up from 1MB to 8MB, demonstrating the effectiveness of the proposed deduplication algorithm. These results are consistent to the previous storage observations. The smaller the writes, the more data objects are segmented. And thus, it takes more time to calculate the hash value for each data object, spends more time to transmit these small data objects to the cluster. When the file is written in the second time (grey line), the data in the current file are duplicate. The client will perform the deduplication on the data, and not write the duplicated data, so the bandwidth is significantly improved.

3) CPU utilization: The CPU utilization in the writing of the file is shown in Figure 11, where the blue line is the change of the client CPU utilization made by the original client, according to the different sizes of the I/O blocks while the other three lines are the CPU utilization changes when the client performs the file write three times with different sizes based on the deduplication client. As can be seen from the figure, the CPU utilization of the deduplication client is higher than that of the original client since the client need to calculate the fingerprint of the data block every time the file is written. Moreover, when using the deduplication client to write the file in the first time, the CPU utilization gradually decreases as the block size increases. This is because when the data block size is large, the same file can be divided into a small number of data objects, and thus it performs less hash calculations. Consequently, although the amount of data is not changed, the number of calculations is less, so the computing resources consumed by the client nodes are decreased. When the duplicated files are written mulitple times, there would be more data and hash objects in the system. Therefore, it requires more calculations when judging the data duplication.

4) Read performance: Now we evaluate the performance of read operation with respect to the proposed deduplication. The bandwidth results are shown in Figure 12. As with the previous experiments, the blue line is the bandwidth change when the original client is used to read the file based on

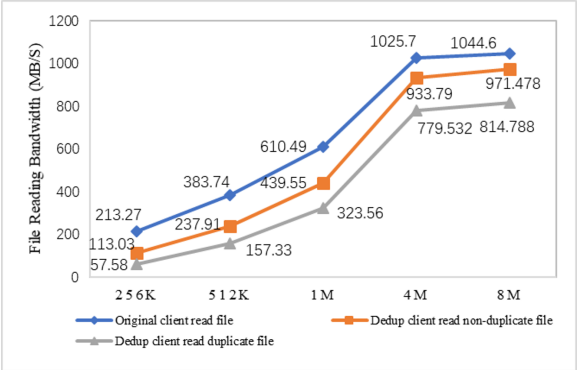

Figure 12: Network bandwidth while reading the duplicate or non-duplicate file.

different I/O block sizes. The brown and grey lines are the bandwidth changes when the deduplication client reads the test file and its deduplicated version in different block sizes, respectively. For all these cases, one can observe that the bandwidths are gradually increased as the block sizes grow up from 256KB to 8MB, the performance of the original client is the best among the three, and the deduplication client is the worst.

These results are not difficult to understand. When the file write is smaller, the bandwidth of the deduplicated client reads the file is lower since when reading the file, each data object needs to first read the extended attribute of the object to determine what the current object is, and then to read the original object again based on the hash value if the current object is a deduplicated object, which leads to a lower bandwidth. As the read size is increased to 4MB, the performance is also improved. This is because the deletion operation in the deduplication client only manipulates the reference counter information in the hash object.

5) Deletion performance: We evaluated the deletion performance by using the deduplication client to write the test file three times in 4MB block size, and then measuring the time and space used when one of them is deleted, and other two are read back. Our experimental results show the deletion cost is comparable to the case in the original client, and the read performance is consistent with that in our previous experiments.

V. Related work

The deduplication algorithm in general follows the same workflow to reduce the redundant data in the storage, which consists of chunking, hashing, indexing, and storage management. In this section, we briefly overview some related work on the data deduplication techniques, especially its design and implementation based on the underlying platform features. A more thorough survey can be referred to [3].

The fixed-size chunking (FSC) is the simplest data partitioning method, whose advantage is simplicity and efficiency. The boundaries of the data block containing the modified part and all subsequent data block would be affected when deletion or insertion occurs in the file, which leads to the follow-up data block is quite different with that all of before. Thus, the FSC recognition rate of duplicate data is significantly reduced. To solve the problem of boundary transfer, content-defined chunking (CDC) was proposed by Quinlan [14] and Muthitacharoen [15], which determine the data block boundary dividing by calculating data fingerprint in the window and matching them, while the sliding window technology is used. The CDC can increase the rate of data deduplication. But QuickSync [16] showed that calculation of the synchronization of cloud storage cost is very high in the data deduplication algorithm based on the CDC. In the process of *Ceph* storing files, the *Ceph* client will package the files with fixed size into objects, and then the objects are stored in the corresponding *OSD* . In order to fit the *Ceph* storage process and not affect the *Ceph* storage efficiency too much, we choose the FSC technique.

The fingerprint technology simplifies the process of identifying the redundant data. In some early data compression methods (such as LZ compression [17] and Xdelta [18]), the weak hash keys and byte-by-byte comparison were used to confirm data duplication. and the fingerprint matching means that the data content may be the same. According to the *birthday paradox*, the collision probability of the SHA-1 algorithm is less than 10^{-20} for the EB-level data set if 8KB block size is used. In the storage system, the probability of a hardware error is $10^{-12} \sim 10^{-15}$, which is several orders of magnitude higher than that of the data collision. The SHA-1 has been used in LBFS, Venti [15] and DDFS deduplication systems [14], [15], [19]. As with these studies, the deduplication algorithm we designed also use the SHA-1 algorithm.

After the data fingerprint algorithm has been determined, the storage mode of data fingerprint should be considered. Venti [15] stored all the block fingerprints index in memory, to quickly and completely identify duplicate data. However, with the explosive growth of data volume, it is difficult to store all the fingerprint index in memory, resulting in frequent access to low-speed disk to query the fingerprint index. Zhu et al. [19] first proposed and utilized the locality of the backup stream in the DDFS system to improve the performance of data deduplication and speed up fingerprint searching. However, the index memory of this method is still very expensive, costing almost $125GB$ memory for $1PB$ backup data. In recent years, some scholars proposed a new architecture, Cache Deduplication [20], based on the flash memory as the main storage, which separates the data cache and the metadata cache for data deduplication. The metadata cache includes a source address index and a fingerprint storage, and the data cache is stored on the flash device. The cache replacement algorithms for duplicate data awareness (such as D-LRU, D-ARC) are proposed for further optimization of the cache space.

Myoungwon et al. [21] proposed a double hashing algorithm to exploit the hashes used by the underlying scale-out storage. A global deduplication framework for shared nothing storage systems has been proposed, which is suitable for the *Ceph* distributed file system [22]. However, these two framework uses database to store data block fingerprint centrally, which has a large cost in fingerprint computing and storage. And there is a single node bottleneck problem for the database node. We designed the data fingerprint object based on the characteristics of *Ceph*, stored the fringerprints of data blocks distributedly in the cluster without any single node bottleneck.

VI. Conclusion

In this paper, we introduced an online deduplication algorithm to *Ceph* distributed storage. The essence of this algorithm is to achieve the deduplicated distributed storage at block level, and the data block is indexed by means of a *hash object* data structure, which is defined for fingerprint storage and redundant data reduction. By using different block sizes to read and write files, we compare the bandwidth, storage space and CPU utilization with respects to file reads and writes, and determine the optimal block size for the deduplication. Our experimental results show that if an appropriate block size is well selected, the use of the deduplication can save up to 99.6% of the storage space, and in the meanwhile, increasing the network bandwidth for write operation more than 2 times, compared to the original *Ceph* in absence of the deduplication. Currently, the proposed deduplication only works for read-only data, fails to update the stored data in place. In the future work, we will address the update operation and apply the algorithm to the *Ceph* object gateway interface to solve the problem of data duplication in the object storage.

Acknowledgment

This work was supported in part by the Research Center for Ecology and Environment of Central Asia, Chinese Academy of Sciences, Shenzhen Basic Research Program (JCYJ20170818153016513), and the CAS Light of West China Program (2016-QNXZ-A-5).

References

[1] A. Sage, A. Scott, L. Ethan, and D. E. Darrell, "Ceph: a scalable, high-performance distributed file system," vol. n/a, pp. 307–320, 2010.

[2] Z. Sun, J. Shen, and J. Yong, "Dedu: Building a deduplication storage system over cloud computing," in *Proceedings of the 2011 15th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*. IEEE, 2011, pp. 348–355.

[3] W. Xia, H. Jiang, D. Feng, F. Douglis, P. Shilane, Y. Hua, M. Fu, Y. Zhang, and Y. Zhou, "A comprehensive study of the past, present, and future of data deduplication," *Proceedings of the IEEE*, vol. 104, no. 9, pp. 1681–1710, Sep. 2016.

[4] M. Chen, Y. Wang, X. Zou, S. Wang, and G. Wu, "A duplicate image deduplication approach via haar wavelet technology," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligence Systems*, vol. 02, Oct 2012, pp. 624–628.

[5] L. Zeng, S. Xu, and Y. Wang, "VMBackup: An efficient framework for online virtual machine image backup and recovery," *Concurr. Comput. : Pract. Exper.*, vol. 28, no. 9, pp. 2630–2643, Jun. 2016.

[6] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in hpc storage systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 7.

[7] A. Prahlad, M. S. Muller, R. Kottomtharayil, S. Kavuri, P. Gokhale, and M. Vijayan, "Data object store and server for a cloud storage environment, including data deduplication and data management across multiple cloud storage sites," 2012, uS Patent 8,285,681.

[8] "FUSE: Fileystem in Userspace," 2017. [Online]. Available: http://sourceforge.net/

[9] A. Rajgarhia and A. Gehani, "Performance and extension of user space file systems," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10, 2010, pp. 206–213.

[10] B. K. R. Vangoor, V. Tarasov, and E. Zadok, "To fuse or not to fuse: Performance of user-space file systems," in *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, ser. FAST'17, 2017, pp. 59–72.

[11] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "Crush: Controlled, scalable, decentralized placement of replicated data," in *SC '06: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, Nov 2006, pp. 31–31.

[12] M. Poat, J. Lauret, and W. Betts, "Posix and object distributed storage systems performance comparison studies with real-life scenarios in an experimental data taking context leveraging openstack swift & ceph," in *Journal of Physics: Conference Series*, vol. 664, no. 4. IOP Publishing, 2015, p. 042031.

[13] "FIO Flexible I/O tester," 2019. [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html

[14] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *FAST*, vol. 2, 2002, pp. 89–101.

[15] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *ACM SIGOPS Operating Systems Review*, vol. 35, no. 5. ACM, 2001, pp. 174–187.

[16] Y. Cui, Z. Lai, X. Wang, and N. Dai, "Quicksync: Improving synchronization efficiency for mobile cloud storage services," *IEEE Transactions on Mobile Computing*, vol. 16, no. 12, pp. 3513–3526, 2017.

[17] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977.

[18] J. MacDonald, "File system support for delta compression," Ph.D. dissertation, Masters thesis. Department of Electrical Engineering and Computer Science , 2000.

[19] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system." in *Fast*, vol. 8, 2008, pp. 1–14.

[20] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication-large scale study and system design," in *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC'12)*, 2012, pp. 285–296.

[21] M. Oh, S. Park, J. Yoon, S. Kim, K.-w. Lee, S. Weil, H. Y. Yeom, and M. Jung, "Design of global data deduplication for a scale-out distributed storage system," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018, pp. 1063–1073.

[22] A. Khan, C.-G. Lee, P. Hamandawana, S. Park, and Y. Kim, "A robust fault-tolerant and scalable cluster-wide deduplication for shared-nothing storage systems," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2018, pp. 87–93.