DedupeSwift: Object-oriented Storage System based on Data Deduplication

Jingwei Ma, Gang Wang✉, Xiaoguang Liu✉
College of Computer and Control Engineering, Nankai University, Tianjin, China
Email: {mjwtom, wgzwp, liuxg}@nbjl.nankai.edu.cn

***Abstract*—Recent years have witnessed the explosion of the data universe. Facing the rapid growth of the data size, cloud storage is proposed as an approach to provide cost-efficient and reliable data storage service. As data size grows, data centers providing cloud storage service need more storage resources to meet the ever-increasing requirements. Data deduplication is a technology aiming to remove redundant data blocks. It has been used to reduce the storage footprint of backup and archival systems. In this paper, we propose DedupeSwift, which is based on OpenStack Swift, an open-source object-oriented storage software widely used in public and private clouds. Data deduplication is introduced to reduce the storage overhead. To deal with the performance overhead brought by deduplication, a lazy method is introduced to reduce the disk I/O bottleneck. Compression and caching are also used in the system to improve the read performance. Experimental results show that our proposed DedupeSwift can reduce the storage overhead by 65.24% and 89.84% on the two data sets with favorable upload and download throughput.**

I. Introduction

Cloud computing [1] has been a hot topic for years. It provides the service in a pay-as-you-go manner, including Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS). The seemingly unlimited virtual resources including network, server, operating system are dynamically managed by the users without knowing the details of the platform and implementation.

Cloud storage arises as one important component in cloud computing. The rapidly developed technologies make it a good way to supply cost-efficient and reliable storage service. With the explosion of the data universe, the demands on storage resources increase exponentially [2], [3], making cloud storage more important. For example, the images in Facebook reached 260 billion in 2010 [4] and took over 20 petabytes. Also, the IDC [3] shows that the universe of data will reach 44 zettabytes by 2020.

Recently, many companies choose to store data in clouds rather than maintain their own storage platforms [5]. This trend makes cloud storage market increase fast. Cloud storage service providers like Amazon Simple Storage Service (Amazon S3) [6], Dropbox [7] offer both highly available and reliable storage at relatively low costs.

Storage clusters, working in the background of cloud storage systems, act as the physical storage platforms. Openstack [8] is a set of open-source software helping people create high performance private or public clouds. Swift [9] is the object-oriented storage component of Openstack. It provides storage service for static data such as virtual machine images, photo storage, email storage, backup and archives.

With the explosive growth of the data volume, to make data management scalable in cloud computing becomes a challenge. Moreover, data redundancy has been revealed in Virtual Machine [10], [11], enterprise [12], [13], [14], [15] and High-Performance Computing [16], [17]. Cloud storage systems have to compress the data in order to reduce the cost.

Data deduplication has been demonstrated to be an effective approach in cloud backup and archival applications to reduce the backup window, also to improve the storage efficiency and network bandwidth utilization. It also has shown the ability to reduce the footprint in the primary storage system. In this paper, we proposed DedupeSwift to remove the redundant data chunks stored in Swift.

Swift has an asymmetric structure, in which the proxy server provides access authorization to the system. On write, the proxy server mainly forwards the data to the object servers. On read, the proxy server fetches data from the object servers and transfers it to the client. The bottleneck is mainly on network and disk I/O. This is a waste of ever-increasing computing power. So we bring deduplication to reduce the storage overhead by utilizing the computing resources.

Because the proxy server just forwards the data without taking a glance at the contents of the data. It wastes a lot of space, ignoring duplicate data blocks. Besides data deduplication, commonly used compression is also combined with data deduplication. However, the slow decompression can negatively affect the read performance. So we use lz4hc [18] to compress the unique data chunks since it has similar compression rate with zlib [19] but the decompression speed is several times faster. In addition, we also use the proxy server as the cache for the data. When reading data from swift, the cache reduces the retrieves from the object servers.

Our contributions include but not limit the follows.

- Propose and implement an exact deduplication based on Opentack Swift object-oriented storage system.
- Eliminate the disk bottleneck by introducing the lazy deduplication method [20].
- Introduce the cache module on the proxy server. With fast decompression, it improves the read performance.
- Present an asynchronous compression scheme, which further reduces the storage overhead with almost no performance penalty.

The rest of the paper is organized as follows. Section II gives the related work. To have a better understanding of the DedupeSwift, an outline of Openstack Swift is shown in Section III. Section IV depicts how deduplication works in DedupeSwift. Compression and Caching are illustrated in Section V. We test the performance of DedupeSwift in Section VI. Section VII summarizes the paper and suggests future work.

II. Related Work

Data deduplication has proven its effectiveness in storage area. It is computationally intensive due to chunking, fingerprinting and compression. When the data size is very large, the main memory is not able to hold all the fingerprints, so fingerprint identification needs many disk accesses (disk bottleneck). As a consequence, data deduplication is also I/O intensive. So researchers improve the deduplication performance either by accelerating the computational tasks or eliminating the fingerprint lookup disk bottleneck.

The calculation overhead can be reduced by using GPU [21], [22] and co-processor [23]. Also, the disk access is reduced by 99% taking use of the locality existing in the data streams [24].

Today, SSDs have shown advantages in random read and sequential operations over HDDs. Many researchers put the fingerprint index on SSDs to improve the fingerprint lookup performance. Due to the limited erasure time, the system needs to avoid random writes. Dedupv1 [25] is designed to take the advantage of the SSD technology. ChunkStash [26] also utilizes the high random read performance to gain improvement of fingerprint lookup. It uses Cuckoo Hashing [27] to resolve the collisions.

Approximate deduplication methods make trade-off between deduplication factor and throughput. Sparse indexing [28], Extreme binning [29] and SiLo [30] are typical approximate deduplication methods, which fetch the fingerprints that are the most likely to be identical with the new ones. As they do not do on disk fingerprint comparison, the throughput is high.

Single node deduplication systems are usually with limited performance, so deduplication cluster is proposed to take use of the parallel computing and I/O resources. The most gain can be obtained when any chunk can, in principle, be compared with any other chunk and be omitted if a match is found. However, such complete matching is harder as the amount of data and components in a large-scale storage system grow [31].

Partial-index strategy improves scalability. When each node holds a part of the entire index, how to route the data becomes the key issue. The locality in the streams again plays an important rule in the routing algorithms.

Extreme binning [29] routes files according to the minimum fingerprint of all the chunks in that file. Different versions of the same file will be likely routed to the same node. Other than file similarity, the locality is also used combined with stateless or stateful routing mechanisms at superchunk granularity to determine which node the chunks should be transferred [32]. Stateless strategy shares nothing and achieves lower space saving while stateful method needs more CPU, RAM and bandwidth. Σ-Dedupe [33] inspired by SiLo, uses handprinting to detect the node with the most potential duplicate chunks.

It is possible to build exact deduplication cluster sharing the fingerprint index. Clements et al. propose a decentralized deduplication based on a SAN [11] cluster aiming to reduce the space footprint of virtual machines. The deduplication runs *out of band* to reduce the negative impact on the system. It achieves 80% space saving with minor performance overhead. Since all the nodes share the fingerprint index via the SAN cluster, it achieves exact deduplication. Debar [34] is designed to use two-phase deduplication scheme exploiting the memory cache and the locality in data streams. It improves the performance of both single deduplication server and distributed implementations. Kaiser also presents an exact deduplication in which all the data is shared via SAN cluster [35]. In that system, the data chunks are not transferred on the network and the bandwidth requirements are low.

Although data deduplication brings a lot of benefits, security and privacy concerns arise. Traditional encryption, while providing data confidentiality, is incompatible with data deduplication. Li proposed several new deduplication constructions supporting authorized deduplication [36]. Also, a new cryptographic method based on the Merkle-based Tree is proposed to solve the Proof of Ownership (PoW) problem [37].

In this paper, we also implement exact deduplication. Instead of storing the metadata on the SAN cluster, we just use the SSD on the proxy server to accelerate the fingerprint identification process.

III. Architecture of OpenStack Swift

A. Logical Structure

There are three levels in Swift to access an object and the logical relationship of them is shown in Figure 1. Each layer has its own type of server to control the access. A user needs an account in order to get access to the objects. Under the account, the user can create containers and each container is able to hold unlimited number of objects. Users use a path like "/account/container/object" to access the corresponding object. Instead of directly using the path to access the object, swift first calculates the hash value of the path with md5 algorithm. Then it searches on the ring with the hash value and determines which object server stores that object.

The primary job of the account server is to handle the task of listing containers. It does not keep the positions of the containers, but the ownership between accounts and containers. Just like how swift determines the position where the object is stored, it uses a ring to determine which account server to connect and get the authorization. Physically, the ownership information is stored in sqlite [38] database. Also, there is some statistical information like the total number of objects, and the usage of the storage space stored.

The container server works like the account server and it is responsible for listings of objects.

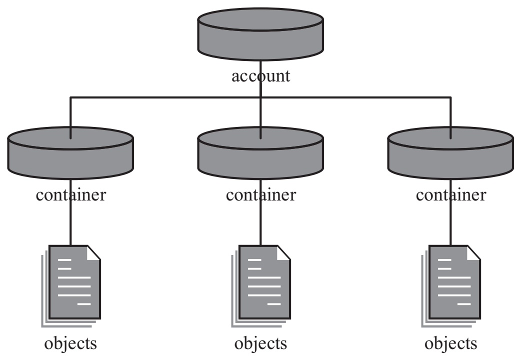

Fig. 1: Logical structure of the path.

The object server works like a blob storage server. It stores, retrieves and deletes objects on local devices. Objects are stored as binary files and the metadata is stored in the extended attributes (xattrs). The physical storage path is determined by hashing the logical path of an object and searching for it on the ring.

B. Data access

Rings play an important role in swift since the proxy server determines which node to send the requests and forward the data by checking them. Each level has their own ring and all the servers in the same level share the same ring. Figure 2 shows an example of the object ring. A server is mapped to one positions on the ring. By hashing the node id, it is expected that different nodes are mapped to different positions and all the nodes are distributed on the ring evenly. A node covers an area between it and the next node either clockwise or anticlockwise. An object is also mapped on the ring within one area (by hashing its path). The node covering that area will store the object.

The account ring and container ring work the same way as the object ring.

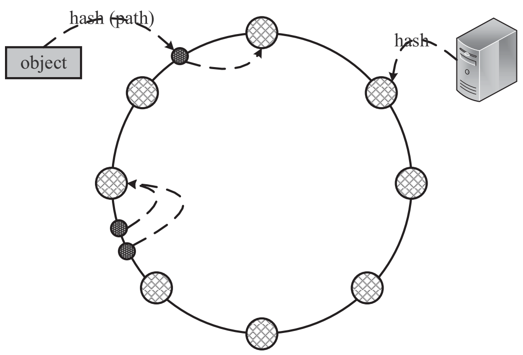

Fig. 2: Ring.

To keep the reliability of the objects, Swift chooses several nodes to store one object. It either uses replication or erasure coding to distribute the data of the object. 3-copies strategy is chosen to store the objects by default, meaning an object are stored in 3 different object servers. To improve the reliability, the chosen nodes usually reside in different zones.

The data stream is shown in Figure 3. The client connects to the proxy server via http. The proxy server calculates the hash of the path in the request. According to the hash value, the proxy server determines which object severs to store the object. Then the proxy server reorganizes the data (coding or copying) and forwards it to the nodes.

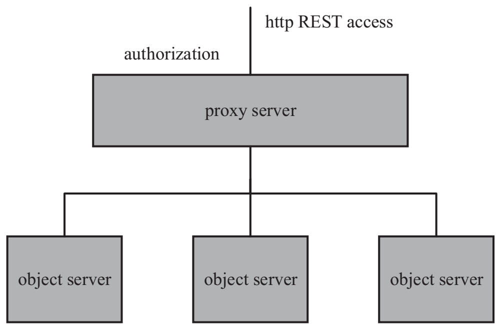

Fig. 3: Data stream.

IV. Deduplication

DedupeSwift is implemented based on Openstack Swift. Therefore, it has the same architecture. In Swift, all the clients wanting to access the data will communicate with the proxy server first to get authorization. Also, all the data goes through the proxy server, which gives the proxy server all the information to reduce storage overhead by compressing the data.

A. Exact Deduplication

Figure 4 shows the process of deduplication in Swift. The original object is segmented into chunks. The chunking method is configurable. Users can choose fixed-size or variable-size chunking method and in the variable-size chunking configurations, the expected size can be adjusted. The system calculates a cryptographic hash as the fingerprint for each chunk. In the system, md5 is used as the hash method. After that, the fingerprint is checked to determine whether the chunk is unique or not. The unique chunk will be put into a chunk container. A chunk container is able to held 4096 chunks by default. When a chunk container is full, it is sealed and compressed to further reduce the storage footprint. Then the compressed chunk container is stored on disk. During the process of deduplication, a file recipe is constructed. It consists of the fingerprints of each chunk in that object.

The main deduplication work is done in the proxy server, including chunking, fingerprint calculation and fingerprint identification. After that, chunk containers are sent to the object servers. File recipes are also sent to the object servers. The object servers do not care about the contents and treat each chunk container or each file recipe as an object.

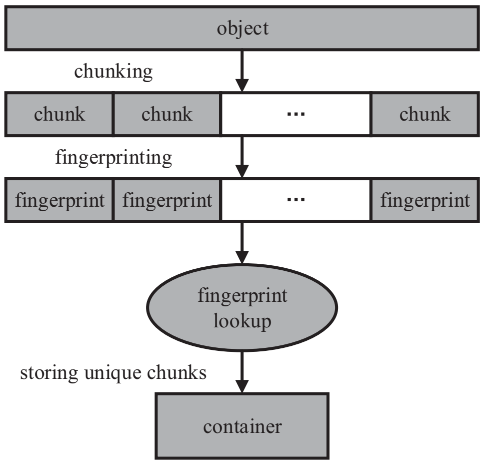

Fig. 4: Deduplication process.

Figure 5 depicts how the proxy server organize the fingerprints and data chunks. As data deduplication searches for fingerprints to find redundant chunks, the proxy server keeps all the fingerprints on its local disk. The size of the fingerprints is much smaller than that of the chunks. So this will not consume much space. To accelerate the fingerprint identification process, the proxy server keeps a fingerprint cache in the memory. LRU eviction policy is used in the cache.

As we store all the chunk containers in the object servers, they are regarded as the chunk store in the deduplication system.

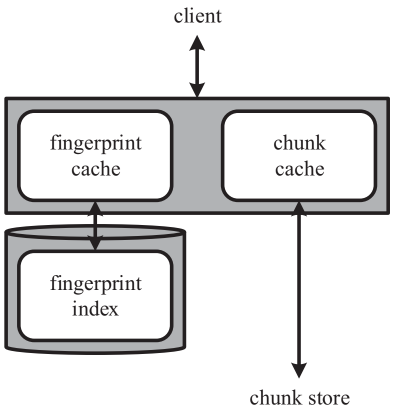

Fig. 5: Proxy server with deduplication.

On read, the proxy server fetches the file recipe of that file. According to the fingerprints in it, the proxy server retrieves the chunks and reconstructs the object. As the chunks are stored in the chunk container, the proxy server reads the entire container from one object server and find the corresponding chunk. The rest chunks in that container will be added into the chunk cache. If the success chunks are found in the cache, the retrieving time is saved for them.

To keep the reliability of the data. Both chunk containers and file recipes are stored in 3 different object servers. The proxy server uses a ring to choose the object servers just as the original swift does.

One concern is that the system may lose the fingerprint index if the proxy server fails. However, the system saves chunk containers on object servers. The chunk containers store all the fingerprints within the corresponding chunks. Once the proxy server fails, the system can reconstruct the index from the containers.

Here, we organize unique chunks into chunk containers. Another way is to send a chunk as the basic unit. In that way, the system does not need to maintain the "fingerprint→container" index. Because it can determine which object server the chunk resides just according to the fingerprint of the chunk and request the object server to see if the chunk has already been there. However, this approach will introduce the fragmentation problem and negatively impact the performance. The proxy server needs to build one connection for each chunk to send or retrieve it. The overhead for these operations will be huge. On the object server side, there is small disk I/O, which will involve many disk accesses and the throughput of the disk can be very low.

Data deduplication introduces disk I/O to the proxy server due to fingerprint identification, which negatively affects the performance. However, swift provides service for large objects. This kind of applications show sensitiveness on the throughput rather than the latency. Traditionally, when a fingerprint is not found in the fingerprint cache, the system will go and search for it on the disk. This means this search process only identifies one fingerprint. The lazy method is based on the observation that several fingerprint lookups need to read the same disk areas to search for the fingerprints in it but at different time points. We buffer the fingerprints rather than search for it immediately.

Figure 6 depicts an example of the lazy method. There are 3 fingerprints mapped to bucket #1. Normally, the system issues 3 disk read operations for these 3 fingerprints. Here, we buffer them until a threshold (3 in the example). Then the system issues one disk read operation and searches for all the corresponding buffered fingerprints. By merging the disk accesses, it largely reduce the disk I/O overhead in data deduplication.

V. Compression and Caching

Traditionally, the object server only store the objects. It uses the ring to determine the position where the object should be stored on the disk. So the disk is busy but the CPU is always idle. Here we introduce compression to the object server, which makes use of the idle CPU resource and reduces the storage overhead. It is possible to compress the container in the proxy server. However, this will consume the limited computing resources on the proxy server.

Since we use 3-replica policy, the total data to be compressed is 3 times more. However, there are much more object servers than proxy server, meaning the total computing power

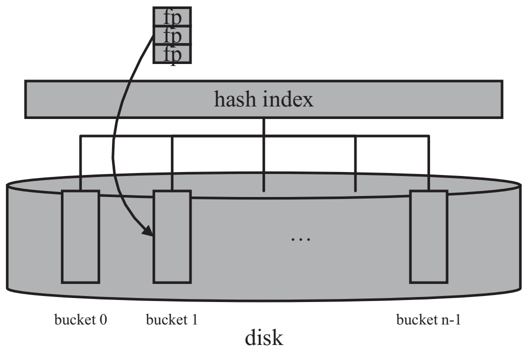

Fig. 6: Lazy method.

on the object servers is several times higher than that of the proxy server. Moreover, compressing the data on the proxy server has to be done as soon as possible to keep the reliability of the data. On the object server, the data can be compressed at any time.

We propose an asynchronous compression strategy to compress the chunk containers and file recipes. From the view of the object servers, they are just objects. Figure 7 describes how the compression works. A compression thread pool and a queue are constructed. On receiving an object, the object server first stores the object on the disk, then it will insert a message in the queue. A compression thread in the pool can pick messages from the queue. Once it gets a message, it will read the object and compress it. After compression, the original object will be replaced by the compressed one. When reads and writes the objects, the compression threads locks the object. But when the object is being compressing, it can be read by other process.

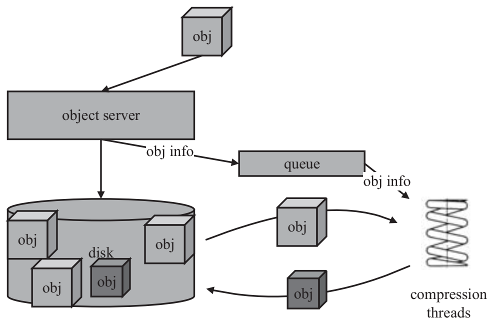

Fig. 7: Asynchronous compression in object server.

DedupeSwift uses a chunk cache to improve the read performance. As the containers are compressed, we divide the chunk cache into two parts. One is used to cache the chunks and the other is used to cache the compressed container. Both of them use LRU eviction policy to replace the items. Due to compression, the cache can hold much more chunks in the compressed sub-cache with the same memory size. By default, the compressed containers take 50% of the total cache space.

Figure 8 shows how the cache works. When a compressed chunk container is retrieved from the object server, it is first put in the compressed part. Then the system decompresses it and puts all the chunks into the uncompressed part. The compressed container remains in the cache until evicted. So it is possible that a chunk resides both the uncompressed part and the compressed part. If the system needs a chunk, it will search for it in the uncompressed part, then the compressed part. Only both cache misses happen in these two parts, the system retrieves the corresponding compressed container from an object server.

Because the cache does not benefit the write process, the upper strategy is only used in the read process.

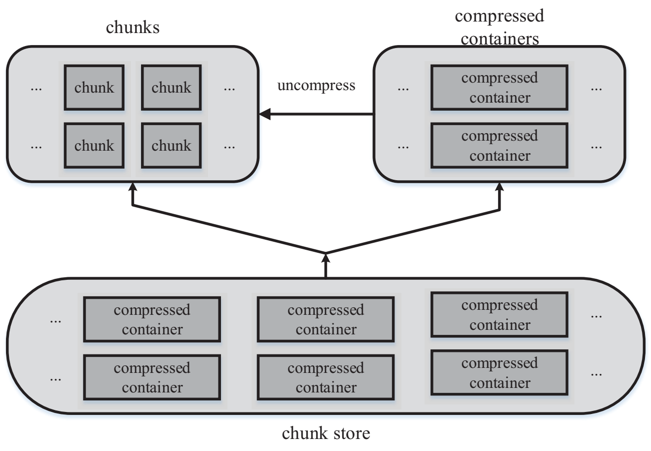

Fig. 8: Chunk Cache.

As the system needs to decompress the data when gets it from the object server or the compressed part of the cache, the speed of decompression seriously affects the read performance. So we choose lz4hc [18] algorithm to compress the data. It has a very high decompression speed, which benefits the read performance.

VI. Evaluation

Table I details the hardware platform used during the tests. All the nodes are connected via the gigabit switch. Since the main bottleneck can be the proxy server. We use a much more powerful node as the proxy server. The fingerprint index is stored on the SSD to accelerate the fingerprint identification process.

TABLE I: Platform.

	proxy server	storage servers and client
CPU	8 × Intel i7 930 @2.80GHz	4 × Intel Xeon @ 3.00GHZ
Disk	Seagate ST2000NM0011 (OS) OCZ-AGILITY3 (dedupe data)	RAID5 HDD 300GB
Memory	6 × 2G	1GB
OS	CentOS release 6.7 2.6.32-573.12.1	
Network	Gigabit switch	

We select 2 data sets in the experiments shown in Table II.

- *Vm* refers to pre-made virtual machine disk images from VMware's Virtual Appliance Marketplace [39], which is used by Jin [10] to explore the effectiveness of deduplication on virtual machine disk images.
- *Kernel* refers to Linux kernel source files [40]. It is widely used in deduplication systems. We choose the versions from 3.9 to 4.5 in the tests.

TABLE II: Data sets.

	total size	file number
Vm	168.42GB	621
Kernel	174.40GB	359

As swift is designed to store large objects, we compact each version of the kernel source code into a file by reading out all the data and appending the index information at last. During the tests, each file is regarded as an object, meaning an uploaded/download operation uploads/downloads the entire file. For the Vm data set, each image file is regarded as an object since the file size is large enough.

We implement DedupeSwift based on Openstack Swift 2.2.2. MD5 is chosen as the fingerprint of each chunk. We use CDC chunking method with 8KB target size on Kernel. During the chunking process, Rabin fingerprint are calculated as the hash value in the sliding window. Since it has been proven in Jin's paper that fixed-size chunking method even has a better deduplication factor than variable-size chunking approach for virtual machine images, we use 4KB fixed-size chunking method for the Vm data set. One chunk container is able to hold 4096 chunks. The fingerprint cache has the capacity to cache 32×2^{20} fingerprints and the chunk cache is set to 8GB.

Python-swiftclient [41] acts as the client to upload and downloads the objects. The client uploads all the files to DedupeSwift then downloads the first 100 files. The tests are focused on the space saving, read performance and write performance.

To eliminate the impact of the cache in the file system, we take the experiments bypass the file system, meaning the proxy server directly read the index from the disk to check the fingerprints.

A. Space Saving

In the system, both deduplication and compression are introduced to reduce the space overhead. Table III reveals how much space saved by each of them separately and also the total space savings combing them. Deduplication alone saves 41.38% and 70.16% of the total space for Vm and Kernel. Compression also can reduce over half of the space overhead. Together, they can reduce 65.24% and 89.84% of the total space. As deduplication generates file recipes, the total space saving is not the product of these two factors. Moreover, since both compression and deduplication reduce the data size by replacing redundant data blocks, there are overlaps between them. This also affects the compression ratio.

Vm does not have as many redundant chunks as Kernel. Also, it has worse compression ratio. So applying data deduplication and compression on it does not get as much space saving as on Kernel. Another reason is it has smaller chunk size, making the file recipes larger than that of Kernel.

TABLE III: Space savings.

	deduplication alone	compression alone	together
Vm	41.38%	64.58%	65.24%
Kernel	70.16%	73.80%	89.84%

B. Asynchronous Compression

To show how the asynchronous compression improves the performance, we test the write performance comparing synchronous compression approaches with the asynchronous strategy. Table IV illustrates the results. "Synchronous proxy" means the chunk containers and file recipes are compressed synchronously on the proxy server and "synchronous object" indicates compression happens synchronously on the object servers. "Asynchronous" is the proposed asynchronous compression which compresses the chunk containers asynchronously on the object servers. Since the proxy server has a higher compression performance, it shows a higher throughput when do synchronous compression. However, it is still slower than the asynchronous compression. Therefore, moving compression to the background effectively reduce the impact of the compression on the write performance.

The system works better on Kernel since it has lager chunk size and the fingerprint identification deals data in a larger granularity. So the fingerprint identification shows smaller effect on the performance.

TABLE IV: Write performance with different compression approaches (MB/s).

	synchronous proxy	synchronous object	asynchronous
Vm	8.91	7.89	10.54
Kernel	17.91	15.79	25.51

C. Advantage of lz4hc

To reveal how lz4hc outperforms other compression algorithm in the system. We compare it with the commonly used zlib compression method. A file from each data set is randomly picked out and compressed. The compression algorithms are directly applied on the raw data without deduplication.

Table V gives the compression/decompression speed and the compression rate $\left(\frac{compressed\ size}{original\ size}\right)$. According to the results, lz4hc and zlib have similar compression rate and compression speed. However, the decompression speed of lz4hc is over 5.5 times faster on the proxy server and over 3.4 times faster on the object server than that of zlib. The extreme fast decompression speed benefits the read process.

TABLE V: Compression/decompression performance on different nodes.

		com. speed (MB/s)		decom. speed (MB/s)		com. rate
		proxy	storage	proxy	storage	
Vm	zlib	19.48	9.65	220.49	132.90	32.56%
	lz4hc	24.31	11.82	1654.73	527.71	37.59%
Kernel	zlib	23.73	11.52	241.17	142.48	21.14%
	lz4hc	27.26	12.64	1331.49	486.61	24.82%

D. Cache and Read

Fragmentation is the natural character of data deduplication. So, this will seriously affect the performance. Here, we depict how the cache and compression together improve the read performance of data deduplication. When needing a chunk, the system checks if it is in the uncompressed part of the cache. On cache miss, the system searches for it in the compressed part of the cache. If the chunk does not exist in the cache, the corresponding chunk container is retrieved and all the chunks in it are inserted into the uncompressed part of the cache while the compressed container itself is added in the compressed part of the cache.

To reveal how the cache help improve the read performance, we give the cache hit rate in Table VI. From the table, we find the cache is really effective since around 99% chunks will be gotten from the cache. The high cache hit rate is due to the spatial locality in the data stream. When one chunk is read, the adjacent chunks will be read with high probability and the adjacent chunks usually stored in the same chunk container. The high cache hit rate saves a large amount of retrieves from the object servers.

TABLE VI: Cache hit rate.

	hit uncom.	hit com.
Vm	99.88%	0.07%
Kernel	98.33%	0.94%

In order to show the advantage of lz4hc compression algorithm, we also use zlib to compress the data and read it out from the server. Table VII shows the results. As the cache hit rate of Vm is really high, we do not see much difference on read performance between the two methods, but lz4hc still outperforms zlib. For kernel, as the cache hit rate is lower, more data needs to be fetched from the object servers or the compressed cache. The advantage is larger.

TABLE VII: Read Performance with different compression algorithms (MB/s).

	lz4hc.	zlib.
Vm	24.65	23.35
Kernel	11.44	9.39

E. Summary

Here we summarize the overall performance and space saving in Table VIII. DedupeSwift is able to save 65.24% and 89.84% space overhead on Vm and Kernel respectively.

The write and read performance is not so high because the tasks introduced by deduplication and compression in the IO path. However, taking space savings into consideration, the performance is still favorable.

TABLE VIII: Summary.

	read (MB/s)	write (MB/s)	space saving
Vm	24.65	10.54	65.24%
Kernel	11.44	25.51	89.84%

VII. Concluding Remarks

In this paper, we propose an object-oriented storage system based on data deduplication. It efficiently reduces the storage overhead by eliminating redundant data chunks. On the two data sets, the storage footprint can be reduced by 65.24% and 89.48%.

By carefully choosing the compression algorithm and introducing cache into the proxy server. The spatial locality of the data streams is utilized and the read performance gets improved.

The read and write performance is still not as good as the original Openstack Swift. Because data deduplication brings overheads like chunking, fingerprint calculation, fingerprint identification and compression. Also, the fragmentation problems seriously affects the read performance. More works can be done to reduce these overheads.

Acknowledgments

This work is partially supported by NSF of China (grant numbers: 61373018, 11301288, 11550110491, 61379146, 61272483), Program for New Century Excellent Talents in University (grant number: NCET130301), the Fundamental Research Funds for the Central Universities (grant number: 65141021) and the Ph.D. Candidate Research Innovation Fund of Nankai University.

References

[1] M. Armbrust, O. Fox, R. Griffith, A. D. Joseph, Y. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, “M.: Above the clouds: a berkeley view of cloud computing,” 2009.

[2] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, “The rise of big data on cloud computing: Review and open research issues,” *Information Systems*, vol. 47, pp. 98–115, 2015.

[3] V. Turner, “The digital universe of opportunities: Rich data and the increasing value of the internet of things. idc iview [electronic resource],” *Access mode: http://www. emc. com/leadership/digital-universe/2014iview/digital-universe-of-opportunities-vernon-turner. htm.*

[4] D. Beaver, S. Kumar, H. C. Li, J. Sobel, P. Vajgel *et al.*, “Finding a needle in haystack: Facebook’s photo storage.” in *OSDI*, vol. 10, 2010, pp. 1–8.

[5] J. Peng, X. Zhang, Z. Lei, B. Zhang, W. Zhang, and Q. Li, “Comparison of several cloud computing platforms,” in *Information Science and Engineering (ISISE), 2009 Second International Symposium on*. IEEE, 2009, pp. 23–27.

[6] “Amazon simple storage service (Amazon S3),” *http://aws.amazon.com/s3/*, 2016.

[7] m. m. m. a. s. r. s. a. p. idilio drago, marco mellia, “Inside dropbox (understanding personal cloud storage services),” *Internet Measurement Conference*, 2012.

[8] "OpenStack," *http://www.openstack.org*, 2016.

[9] "OpenStack Swift," *http://swift.openstack.org*, 2016.

[10] e. l. m. keren jin, ethan l miller, "The effectiveness of deduplication on virtual machine disk images," 2009.

[11] A. T. Clements, I. Ahmad, M. Vilayannur, J. Li *et al.*, "Decentralized deduplication in san cluster file systems." in *USENIX annual technical conference*, 2009, pp. 101–114.

[12] R. Koller and R. Rangaswami, "I/O Deduplication: Utilizing content similarity to improve I/O performance," *ACM Transactions on Storage (TOS)*, vol. 6, no. 3, pp. 1–26, 2010.

[13] D. Meyer and W. Bolosky, "A Study of Practical Deduplication," *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST)*, pp. 1–13, 2011.

[14] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "idedup: Latency-aware, inline data deduplication for primary storage," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association, 2012, pp. 24–24.

[15] A. El-Shimi, R. Kalach, A. Kumar, A. Ottean, J. Li, and S. Sengupta, "Primary data deduplication-large scale study and system design." in *USENIX Annual Technical Conference*, 2012, pp. 285–296.

[16] S. Al-Kiswany, M. Ripeanu, S. S. Vazhkudai, and A. Gharaibeh, "stdchk: A checkpoint storage system for desktop grid computing," in *Distributed Computing Systems, 2008. ICDCS'08. The 28th International Conference on*. IEEE, 2008, pp. 613–624.

[17] 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.

[18] "lz4," *http://www.lz4.org/*, 2016.

[19] "zlib," *http://www.zlib.net/*, 2016.

[20] J. Ma, R. J. Stones, Y. Ma, J. Wang, J. Ren, G. Wang, and X. Liu, "Lazy exact deduplication," in *Mass Storage Systems and Technologies (MSST), 2016 IEEE 32th Symposium on*. IEEE, 2016, pp. 1–10.

[21] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: Gpu-accelerated incremental storage and computation," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST)*, 2012.

[22] X. Li and D. Lilja, "A Highly Parallel GPU-based Hash Accelerator for a Data Deduplication System," in *Parallel and Distributed Computing and Systems*. ACTA Press, 2009.

[23] L. Ma, C. Zhen, B. Zhao, J. Ma, G. Wang, and X. Liu, "Towards Fast Deduplication Using Low Energy Coprocessor," in *2010 Fifth International Conference on Networking, Architecture and Storage (NAS)*. IEEE, 2010, pp. 395–402.

[24] B. Zhu, K. Li, and H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*. USENIX Association, 2008, pp. 269–282.

[25] D. Meister and A. Brinkmann, "dedupv1: Improving deduplication throughput using solid state drives (ssd)," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–6.

[26] B. Debnath, S. Sengupta, and J. Li, "ChunkStash: Speeding up Inline Storage Deduplication using Flash Memory," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIX Association, 2010, pp. 16–16.

[27] R. Pagh and F. Rodler, "Cuckoo Hashing," *Journal of Algorithms*, vol. 51, no. 2, pp. 122–144, 2004.

[28] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, "Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality," in *Proccedings of the 7th Conference on File and Storage Technologies (FAST)*. USENIX Association, 2009, pp. 111–123.

[29] D. Bhagwat, K. Eshghi, D. Long, and M. Lillibridge, "Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup," in *IEEE 2009 International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2009, pp. 1–9.

[30] W. Xia, H. Jiang, D. Feng, and H. Yu, "Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput," in *Proceedings of the 2011 USENIX conference on USENIX Annual Technical Conference*. USENIX Association, 2011, pp. 26–28.

[31] J. Paulo and J. Pereira, "A survey and classification of storage deduplication systems," *ACM Computing Surveys (CSUR)*, vol. 47, no. 1, p. 11, 2014.

[32] W. Dong, F. Douglis, K. Li, R. H. Patterson, S. Reddy, and P. Shilane, "Tradeoffs in scalable data routing for deduplication clusters." in *FAST*, vol. 11, 2011, pp. 15–29.

[33] N. X. Yinjin Fu, Hong Jiang, "A scalable inline cluster deduplication framework for big data protection," 2012.

[34] T. Yang, H. Jiang, D. Feng, Z. Niu, K. Zhou, and Y. Wan, "Debar: A scalable high-performance de-duplication storage system for backup and archiving," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–12.

[35] J. Kaiser, D. Meister, A. Brinkmann, and S. Effert, "Design of an exact data deduplication cluster," in *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*. IEEE, 2012, pp. 1–12.

[36] J. Li, Y. K. Li, X. Chen, P. P. Lee, and W. Lou, "A hybrid cloud approach for secure authorized deduplication," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 5, pp. 1206–1216, 2015.

[37] m. l. nesrine kaaniche, "A secure client side deduplication scheme in cloud storage environments," *New Technologies, Mobility and Security*, 2014.

[38] "sqlite," *http://www.sqlite.org/*, 2016.

[39] "VMWare Images," *http://www.thoughtpolice.co.uk/vmware/*, 2013.

[40] "Linux Kernel," *https://www.kernel.org/*, 2016.

[41] "Python SwiftClient," *https://github.com/openstack/python-swiftclient*, 2016.