2016 IEEE 22nd International Conference on Parallel and Distributed Systems

Zero-Chunk: An Efficient Cache Algorithm to Accelerate the I/O Processing of Data Deduplication

Hongyuan Gao[1], Chentao Wu[1*], Jie Li[1,2], and Minyi Guo[1]
[1]Shanghai Key Laboratory of Scalable Computing and Systems,
Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
[2]Department of Computer Science, University of Tsukuba, Tsukuba, Ibaraki, Japan
*Corresponding Author: wuct@cs.sjtu.edu.cn

***Abstract*—Data deduplication is a technique to eliminate duplicated copies of data. It can save the storage space, reduce the amount of disk I/Os, then improve the system performance. There have been several popular deduplication algorithms such as SISL [30], Extreme Binning [1], Sparse Indexing [14], etc. These schemes use containers to aggregate data chunks for better performance. However, they either suffer from low cache hit ratios or inefficient cache utilization.**

To address this problem, we design Zero-Chunk, a new cache algorithm that balances the cache hit ratio and memory usage. In our method, we choose chunks whose fingerprints have all-zero remainders as pointers (called zero chunks), and aggregate the following chunks into their corresponding containers. And then, when the access patterns change, our method can eliminate cold data chunks and containers to maintain a low overhead. To demonstrate the effectiveness of Zero-Chunk, we conduct several simulations. The results show that, compared to Sparse Indexing (the most popular implementation method in data deduplication), Zero-Chunk improves the cache hit ratio by up to 5.2%, saves the memory consumption by more than 50.7%, and decreases the total number of I/Os by up to 17.3%, respectively.

***Index Terms*—Cache; Deduplication; Backup Systems; Performance Evaluation**

I. Introduction

Nowadays, the total amount of digital bits grow 40% every year, and will reach 44ZB by 2020 [8]. However, a large portion of files share common data with each other, which causes a huge storage waste [26]. One effective method to solve the problem is data deduplication. It saves only one unique copy of duplicated data, and largely relieves the pressure of storage systems.

The most popular way to process duplicated data is chunk-based deduplication [19] [21] [9]. When comparing two chunks, fingerprints, which can be calculated by hash algorithms such as MD5 [23], are usually used to reduce the cost. For example, a 128-bit fingerprint only occupies 1/256 of the space compared to the original 4KB chunk. Since it is costly to find duplicated data chunks in the disk, we sample and cache a small portion of chunks, while these chunks are the most likely to be duplicated by the incoming I/O requests.

There have been many cache algorithms that do a tradeoff to achieve a relatively higher deduplication ratio [11] [29] without sacrificing too much memory space. SISL [30], Extreme Binning [1], Sparse Indexing [14] and SiLo [28] focus on backup systems, where write requests are more than read requests. It requires a higher deduplication ratio and write efficiency. Primary storage systems include daily use, containing both write and read. iDedup [25] and POD [15] are methods for this scenario. Dedupv1 [16], chunkStash [7] and Ordermergededup [6] are solutions for high-performance systems based on solid-state drives (SSDs). SSDs have faster random access and lower read latency, but can suffer from write amplification, so it is important to reduce the number of write I/Os.

In order to achieve a high deduplication ratio with a limited number of sampled chunks, the cache algorithm must be efficient. However, for backup systems, existing methods have several limitations. First, in SISL [30], a Bloom filter [2] is used to eliminate chunks which are impossible to be duplicated, but the filter is less effective if the duplication ratio is high. It is because most chunks have their own copies and pass the filter directly. On the other hand, if the cache hit ratio is low, the looking up processes in the disk are time-consuming. Second, for Extreme Binning approach [1], Chunks are grouped into containers by files, and the chunks with the smallest fingerprint are sampled to point to the container. In this scenario, many files can share a same chunk, and it's not necessary for incoming chunks to find a proper container. Third, Sparse Indexing [14] uses manifests to choose a group of segments (a segment can be viewed as a sequence of chunks). Nevertheless, to duplicate a segment, many segments have to be loaded into the memory, making the cache usage inefficient.

To address the above problems, we propose Zero-Chunk, a new method for chunk grouping and cache replacement. We choose a sampling parameter N and pick chunks whose last N digits of fingerprints are all zero. They are called Zero Chunks (ZCs). Whenever we have a new ZC, the following chunks are recorded and put into the container pointed by this ZC. Therefore, the locality of the cache is based on

1521-9097/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPADS.2016.87

access patterns, which is more reliable and flexible.

We have the following contributions in this paper,

- We use sampled index and maintain a high deduplication ratio by grouping chunks with zero fingerprints.
- We reduce the memory usage by eliminating cold chunks and containers, and select proper parameters to balance the deduplication ratio and the memory usage.
- We implement a sequential containers, which reduces the overhead of the disk.

The rest of this paper is organized as follows. We provide background and motivation are presented in Section II. In Section III, We describe the design and details of our algorithm. In Section IV, We evaluate the performance of the algorithm. Finally, we conclude this paper in Section VI.

II. Background and Motivation

In this section, we discuss the background of existing cache algorithms for deduplication applications and their limitations. We also introduce our motivation for Zero-Chunk.

A. Overview of Data Deduplication

Data deduplication is a technique that stores one copy of multiple same data chunks. It creates a map between the reference of duplicated chunks and their unique locations in the disk. According to different methods to process duplicated chunks, the techniques of deduplication can be divided into the following categories.

1) File vs. Sub-file: File-level deduplication [3] is easy to carry out. Duplicated files can be detected when a hash collision happens, and file-level fragmentation is not widely spread [17]. However, the duplication ratio is relatively low since two files are not viewed as duplicated even if only one bit is different.

Compared to file-level deduplication, sub-file-level deduplication receives more attention [30] [1] [14] [4] [28] [18] [24]. The basic unit of data is called a chunk. According to different methods, data chunks can be fixed-sized of variable-sized. Fixed-sized chunks are easy to be processed and managed. However, if a bit is added or removed at the head of a file, all of the following chunks are changed. Rabin Fingerprints [22] use variable-sized chunks to solve the problem, but it has some difficulties in chunking and management.

2) In-line vs. Post-process: In-line deduplication requires chunking data and calculating fingerprints as soon as new I/O requests come. Since the calculation is costly, the write throughput can be negatively affected and become a bottleneck if the processor is not fast enough. Sometimes, an external device is used to assist the work of CPU. [12].

Post processing just stores all new data temporarily, then processes them when the processor is idle. Therefore, the hash calculation and duplication processes are spared. However, duplicated data can be unnecessarily stored in the disk, which is a huge overhead.

B. A Typical In-line Sub-file Deduplication Workflow

We provide a typical in-line, sub-file deduplication scheme in Figure 1. It can be divided into three procedures.

1) Chunk Data: As I/O requests come into the chunking module, files are broken into fixed-sized chunks, and the size of chunks is the minimum granularity for deduplication. Afterwards, we calculate their fingerprints and use them in the next procedure.

2) Deduplicate chunks: We compare the new fingerprints with existing ones in the memory. Each new fingerprint is assigned with a corresponding sampled chunk, which points to a container in the disk. The container is supposed be highly possible to have duplicated chunks with the those in I/O requests, so it is loaded into the memory. Actually, we expect that the container has been loaded previously, so we can directly begin searching for potential duplicated chunks in the container.

3) Write in or Update: If duplication happens, it's not necessary to write the new chunk into the disk, so we just update the map table to indicate where the new chunk is located. Otherwise, we access the disk and store the chunk.

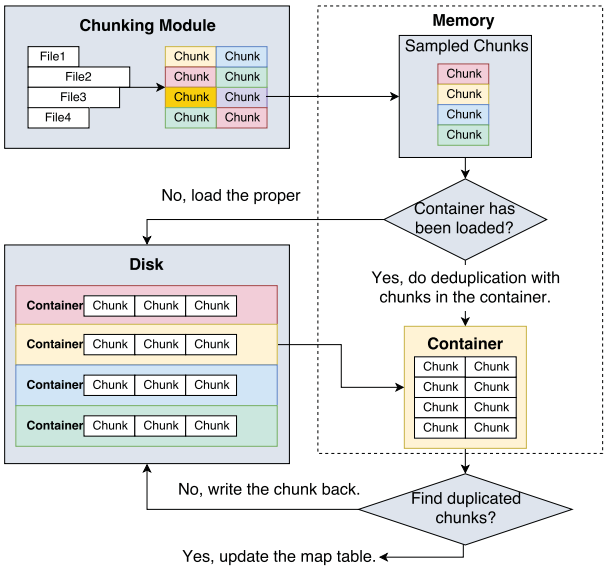


Fig. 1: The work flow of a typical deduplication scheme

C. Existing Cache Algorithms

Using sampled chunks and containers saves the memory consumption, but the cache hit ratio is also reduced. There have been several algorithms that solve the hit ratio problem, and their strengths and weaknesses are shown in Table I.

The scheme proposed by Zhu *et al.* [30] uses a Bloom filter as a pre-process module to avoid unnecessary fingerprint lookups. It eliminates chunks that are impossible to be duplicated, and can save lots of accessing time if the deduplication ratio is low. Nevertheless, if most of

the chunks are duplicated, the Bloom filter becomes less effective because they can directly pass through the filter.

Extreme Binning [1] utilizes files to arrange containers. When a file is processed and chunked, the data chunk with the least fingerprint value is recorded and stored in the memory. Thus, the two files whose least fingerprints are equal are highly possible to be the same. If it fails to find such a collision, then the file is not further duplicated and processed, which can negatively affect the deduplication ratio.

Sparse Indexing [14] uses segments, which are viewed as the most similar to incoming data. It chooses one segment at a time, until it reaches the maximum allowable number or candidate manifests are used up. The similarity makes the chunks loaded in the memory are likely to be duplicated, but it also causes more overhead since segments are chosen by looking up manifests and several segments have to be loaded.

D. Our Motivation

We summarize the existing cache algorithms for data deduplication in Table I. It is clear that an effective algorithm should satisfy three requirements, efficiently utilizing memory space, high deduplication ratio, and low lookup overhead. However, existing methods have several limitations on these aspects (introduced in Section II.C), which motivates us to propose a cache method called "Zero-Chunk".

III. Zero-Chunk Cache Algorithm

In this section, we introduce the Zero-Chunk cache algorithm in detail. Its purpose is to utilize the memory storage more efficiently. To clearly illustrate our approach, we define two types of data chunks,

- **Zero Chunk (ZC)**: A data chunk whose last few bits of its fingerprint are all zero.
- **Non-Zero Chunk (NZC)**: A data chunk that is not a ZC.

A. Overview of Zero-Chunk

The approach of Zero-Chunk is in Figure 2. It can be completed in four steps,

Step 1 (Classify Chunks): The Zero Judger checks whether the remainder of the chunk is zero. If so, the chunk is a ZC. Otherwise, it is a NZC. The chunk is processed either by Step 2 or Step 3 according to its category.

Step 2 (Process ZCs): A ZC is sent to the Zero Table. It checks whether the chunk is duplicated.

TABLE I: Comparison of cache schemes in data deduplication

Schemes	Deduplication Ratio	Memory Usage	Lookup Overhead
Zhu *et al.*'s scheme	low	moderate	moderate
Extreme Binning	moderate	low	high
Sparse Indexing	high	moderate	moderate
Zero-Chunk	high	low	low

Step 3 (Process NZCs): For a NZC, we look for its duplicated chunk in the container which has been loaded into the memory.

Step 4 (Eliminate Cold Chunks): The Eliminator module checks whether the hotness values drop to zero. If so, it eliminates cold chunks and rebuilds the container.

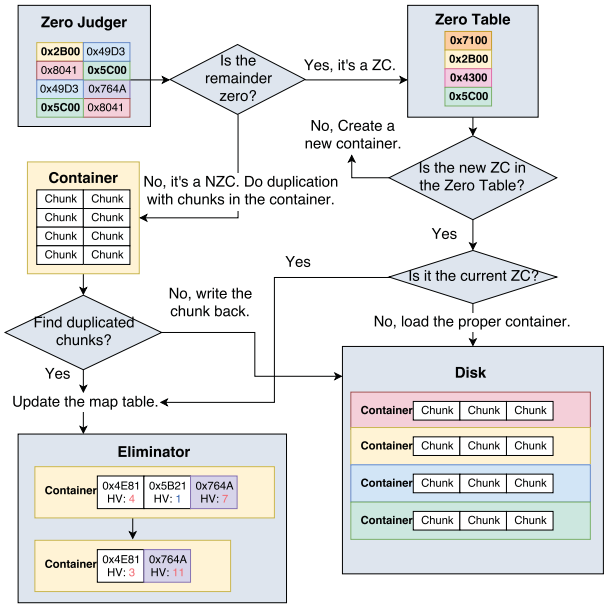


Fig. 2: The workflow of Zero-Chunk cache algorithm

B. Process ZCs

In Zero-Chunk cache algorithm, we define the sampling parameter as N. Thus, a ZC is a chunk whose last N digits of fingerprint is zero. A good fingerprint calculation method is necessary because it ensures that fingerprints are totally random and the proportion of ZCs is very close to the ideal sampling rate, $1/2^N$. In our experiment (see Section IV), we use MD5 hash algorithm to calculate fingerprints. On the other hand, a flawed hash function has the risk causing data corruption if two different chunks have the same fingerprints and one is discarded. For massive amounts of data, we can use complex hash functions with longer hash values such as SHA-1, to minimize the possibility of corruption. It can be even lower than undetected hardware errors. [20]

We maintain a Zero Table to store all ZCs in the memory, unless they are eliminated because of low hotness values. By choosing an appropriate sampling rate, we can keep the storage space of the table very small. The ideal space occupied in the memory can be calculated as the following equation:

$$memory\_space = \frac{hash\_value\_size}{chunk\_size} * sampling\_rate$$

Given the 128-bit MD5 hash function and 4KB chunk size, the relationship between sampling rate and memory

usage per TB of different workloads is shown in Figure 5 on page 6, which is acceptable.

The algorithm of the processing is shown in Algorithm 1. If a new ZC comes, its fingerprint is compared with existing ZCs in the Zero Table, examining whether there has been a duplicated ZC. If the chunk doesn't appear before, we just insert it into the table, and create a new container. Simultaneously, the ZC chunk is assigned a default hotness value, recording the frequency it is accessed. On the other hand, since it is a new ZC, all other zero chunks are not accessed at the moment. Thus, their hotness values are all reduced by 1, marking that they become "colder".

The container is implemented in C in the form of dynamic array. It records the following information,

- Fingerprints of all chunks
- Hotness values of all chunks
- Number of current chunks in the container
- Number of cold chunks in the container ("cold" means that its hotness value is zero)
- Maximum volume of the container

If a duplicated ZC is found in the Zero Table, its hotness value increases largely, while other ZCs' hotness values are reduced. Because of locality, it is possible that a large portion of incoming data chunks following the ZC are the same as those existing in the container. Thus, we load the whole container into the memory.

Algorithm 1: The algorithm for processing ZCs.

```
if chunk.fp << (len_hash - N) == 0 then
    for zc in zeroTable do
        if chunk.fp == zc.fp then
            map.update()
            zc.hv += ht
            if zc != prevZc then
                write(prevZc.address)
                load(zc.address)
            end
        else
            zc.hv -= 1
        end
    end
    coldChunkNum.update()
    if CacheMiss then
        zeroTable.insert(chunk.fp)
        writeData(chunk)
    end
end
```

C. Process NZCs

Actually, most fingerprints don't end with a string of zeroes. For these NZCs, we do deduplication in the container in the memory, which is loaded previously. These new chunks are expected to be duplicated with those in the container.

The algorithm for NZCs is shown in Algorithm 2. If a duplicated chunk is found, the map table is updated and its corresponding hotness value increases, showing that the chunk has been recently accessed. After that, the scanning process continues and decreases the hotness values of other chunks for potential elimination. If a cache miss occurs, the new chunk is added into the container with all hotness values reduced, and the number of current chunks in the container is also updated. Besides, during the process of scanning, the container updates the number of cold chunks immediately so that it can know whether the container has become dirty enough and needs to be cleaned.

Figure 3 shows a circumstance that Zero-Chunk algorithm outperforms Least Recently Used (LRU) policy. The memory can cache four chunks. Under LRU, the hit ratio is 3/10. For Zero-Chunk, we choose 2 as the sampling parameter, so the first chunk 4 is a ZC. Thus, 2 and 3, the two NZCs following 4, are cached by the container of 4. Though the container is cleared because a new ZC 0 comes, it is loaded again when another 4 (the 8th chunk) is accessed. Therefore, 2 and 3 can be hit by the 9th and 10th chunk.

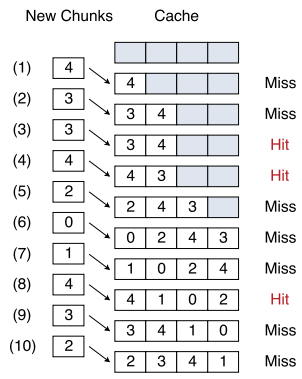
Algorithm 2: The algorithm for processing other chunks.

```
if chunk.fp << (len_hash - N) == 0 then
    for nzc in cache do
        if chunk.fp == nzc.fp then
            map.update()
            nzc.hv += ht
        else
            nzc.hv -= 1
        end
    end
    coldChunkNum.update()
    if CacheMiss then
        cache.insert(chunk.fp)
        writeData(chunk)
    end
end
```

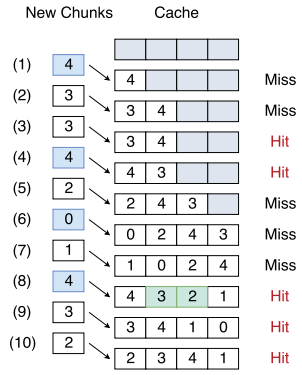
D. Eliminate cold chunks

Because of locality, data chunks from new I/O requests are expected to have their duplicated copies in the memory, but not all chunks in the container are frequently accessed after they are put in. These chunks occupy a large amount of storage space in the cache, and increase the overhead of searching for duplicated chunks. Moreover, some ZCs appear rarely in the requests, making its container unlikely to be loaded again. Thus, the Zero Table is polluted by too many cold ZCs.

To erase cold chunks in a container, we check their hotness values and calculate the number of cold chunks after a new NZC is processed. Since hotness values are updated as the chunks are scanned, it doesn't cause extra overhead. If the number of cold chunks is more than half of the total amount of chunks in the container, a new dynamic array is created and only hot chunks are put into it. The new size is the double of hot chunks. It ensures sequential



(a) LRU



(b) Zero-Chunk

Fig. 3: An example where Zero-Chunk has a higher cache hit ratio than LRU.

scanning of the container, avoiding fragmentation due to the change of chunks. The elimination can be also triggered when the container is full. Since the maximum volume has been recorded, we can know if a larger container should be created. Only hot chunks are moved into the new container, and its new size is also the double of hot chunks. However, doubling a container doesn't occur often. When we scan a container, the location of the first cold chunk is recorded. If a new unique chunk comes, it takes the place of the recorded chunk, instead of being attached at the end of the container. That reduces the frequency of rebuilding containers.

The number of cold containers are checked when a ZC is processed. As ZCs in the Zero Table are scanned, their hotness values are updated. This procedure is like eliminating NZCs. If more than half of ZCs are marked cold, a new Zero Table is created with only hot ZCs. Similarly, cold ZCs and their containers are cleared when the Zero Table is full, and they can also be replaced by newly arrived ZCs.

IV. Performance Evaluation

In this section, we evaluate the cache hit ratio, memory usage and I/O performances of Zero-Chunk compared to other approaches, including LRU, FIFO, LFU and Sparse Indexing [14], to show its advantages on these aspects.

A. Evaluation Methodology

The property of our environment is shown in Table II. To evaluate the disk performance, we use DiskSim 4.0 [5] as the simulator, and choose the *Barracuda* model. We implement FIFO, LRU and LFU in C, and compile them with GCC 4.2.1; For Sparse Indexing, we simulate it with Destor [10].

We use three traces from IODedup [13] and SRCMap [27] collected by Florida International University.

- **Homes-21**: is a subtrace of *Homes*, which includes Research group activities such as developing, testing, experiments, technical writing and plotting.
- **Web**: is made up of CS department webmail proxy and online course management.
- **Web Research**: contains web-based management of 10 FIU research projects using Apache web server.

Each of the trace is a set of MD5 hash values, instead of raw data. It includes write and read requests collected by Florida International University. The properties of each request are arriving time, process ID, process name, logical block number, operation (write or read), major and minor device number. The specifications of the traces are shown in Table III.

TABLE II: Evaluation environment

Item	Description
CPU	Intel Core i7 2.2GHz
RAM	16GB DDR3 1600
Disk	256GB SSD
Language	C
Compiler	GNU Compiler Collection (GCC) 4.2.1
Other Tools	DiskSim 4.0, Destor

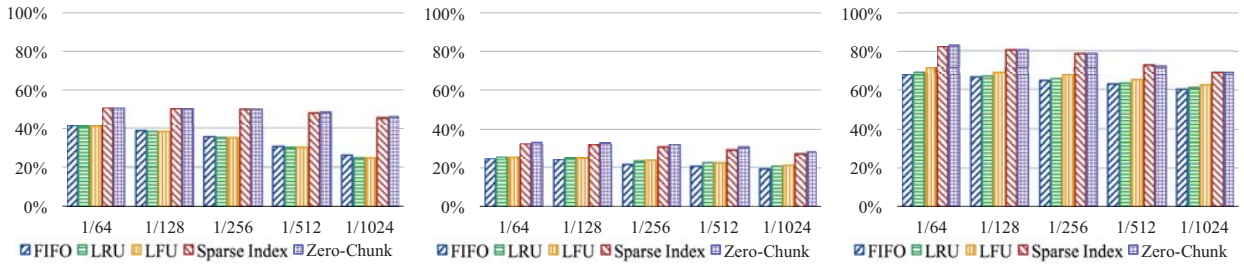
TABLE III: Specifications of the traces

Name	Size	Write Req.	Read Req.
Homes-21	1.94GB	507136	1616
Web	54.53GB	11177702	3116456
Web Research	1.62GB	424512	88

The hotness value helps to control both the number of ZCs in the Zero Table and NZCs in containers. For instance, we set the sampling parameter N as 8, then the sampling rate is 1/256. Ideally, the average number of NZCs between two ZCs in I/O requests is 255. In other words, only about 255 chunks could be accessed after their container is loaded. Thus, to maintain balanced total hotness values, every chunk is assigned an initial hotness value of 255, and the value increases by 255 if the chunk is accessed.

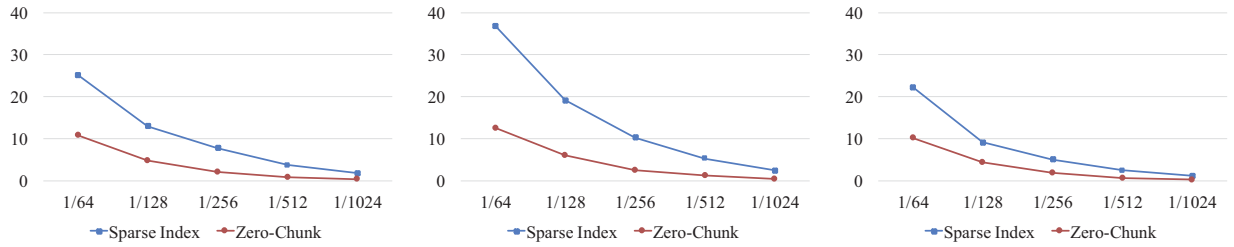
B. Experimental Results

1) Cache Hit Ratio: Figure 4 demonstrates the cache hit ratio under the three traces and different sampling rates. In all traces, Zero-Chunk largely outperforms the three classic



(a) Homes-21 (b) Web (c) Web Research

Fig. 4: Comparison on the cache hit ratio under different sampling rates and traces (the digitals in X-axis are different sampling rates).



(a) Homes-21 (b) Web (c) Web Research

Fig. 5: The memory usage of Zero-Chunk compared to Sparse Indexing. The segment size of Sparse Indexing is 5MB (the digitals in X-axis are different sampling rates).

algorithms, FIFO, LRU and LFU. Its result is also a little better than that of Sparse Indexing.

For *Homes-21* and *Web*, compared to classic algorithms, the superiority of Zero-Chunk varies from 22.4% to 87.7%, depending on the sampling rate. It also increases the hit ratio by up to 4.9% compared to Sparse Indexing.

For *Web Research*, traditional algorithms work rather well, achieving a cache hit ratio higher than 60% even under the 1/1024 sampling rate. That is because one chunk appears many times in the trace. We reckon that it is a chunk whose content is all zero, and calculate the MD5 value of the 4KB file with full of zeroes. The result validates our prediction. These empty chunks appear so often that they are very likely to be found in the cache even using classic algorithms. Zero-Chunk outperforms Sparse Indexing under all sampling rates except 1/512, because Sparse Index uses manifests to get more information for deduplication.

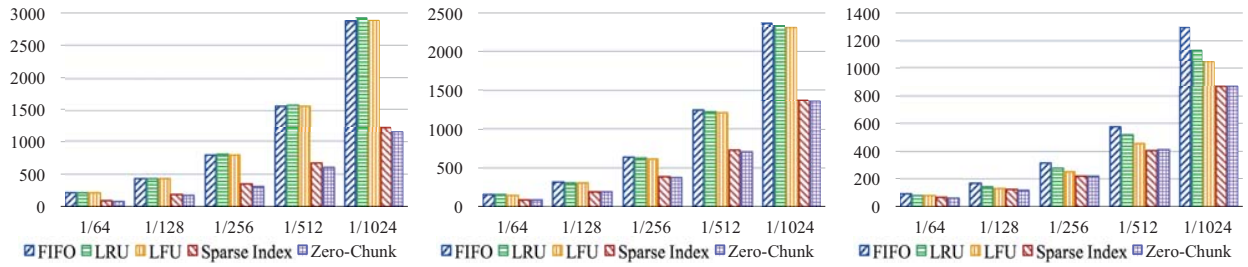
2) Memory Usage: Compared with Sparse Indexing, a main advantage of Zero-Chunk is that it uses less memory for the same deduplication ratio. In our approach, cold chunks and containers are immediately cleared, and the size of containers can also be controlled.

In this experiment, We use the three traces to test the memory usage of Zero-Chunk and Sparse Indexing with different sampling rates. Zero-Chunk only uses up to 49.3% of the memory space compared to Sparse Indexing.

3) Number of I/Os: A well-designed cache replacement algorithm maximizes the usage of memory. It also helps to reduce negative effects of disk I/Os, which largely slows down the system. Zero-Chunk is an inline deduplication solution, so duplicated data chunks are discarded in the memory directly, without written in the disk. Thus, it's necessary to access the disk only when the container is written back.

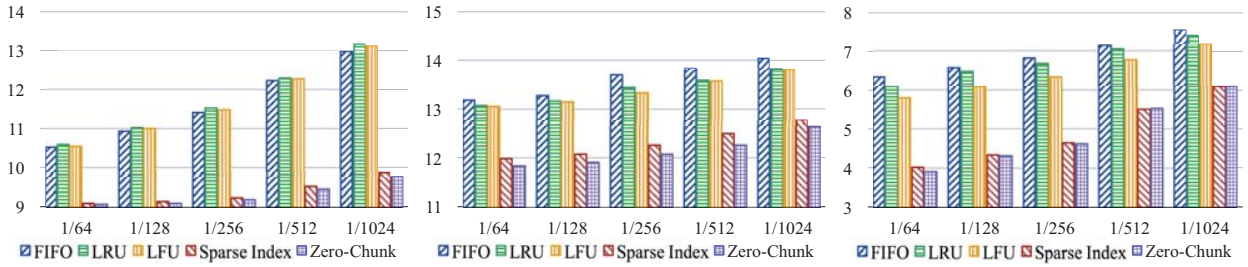
Figure 6 shows the comparison of the number of I/Os. Zero-Chunk shows better results just as in the cache hit ratio experiment. Compared to the traditional policies, Zero-Chunk reduces at most 64.1% of I/Os. Zero-Chunk also outperforms Sparse Indexing by up to 17.3% in three of the traces.

4) Average Response Time: To test the average response time, we modify the traces to simulate the overhead of write and read operations in the disk. When a new ZC comes, the current container is written back to the disk, so we add write operations to the traces and test them in DiskSim. Similarly, if a new container is loaded in the memory, all chunks of the container are added to the traces in the form of read operations.



(a) Homes-21 (b) Web (c) Web Research

Fig. 6: Comparison on the I/O numbers per GB data for the three traces under different sampling rates (the digitals in X-axis are different sampling rates).



(a) Homes-21 (b) Web (c) Web Research

Fig. 7: Comparison on the average response time (ms) using different sampling rates and traces under the disk model of *Barracuda* [5] (the digitals in X-axis are different sampling rates).

Figure 7 demonstrates the average response time under *Barracuda* model of DiskSim 4.0 [5]. Under *Homes-21* and *Web Research*, Zero-Chunk reduces more than 13.9% of the response time compared to classic algorithms. For *Web*, the difference is not so much (8.4% at least) because the deduplication ratio is relatively low in this trace, and all the algorithms need numerous I/O operations. Compared to Sparse Indexing, Zero-Chunks saves the overhead of loading champions, reducing up to 2.9% of the average response time.

C. Analysis

Compared to other popular cache algorithms, Zero-Chunk has high performance gains, which are shown in Table IV and V. There are several reasons to achieve these advantages. First, Zero-Chunk collects the history information of access patterns, which improves the accuracy of prediction. Second, it dynamically eliminates cold data chunks and keeps the cache efficient. Third, Taking the advantage of the hotness value, Zero-Chunk can achieve a balance between deduplication ratio and memory usage, making the algorithm suitable for various deduplication applications.

V. Conclusions

In this paper, we propose Zero-Chunk, a cache algorithm for deduplication in backup systems. Our experiments show that Zero-Chunk has the following advantages compared to other data deduplication schemes: 1) increases the cache hit ratio by up to 5.2%; 2) eliminates cold data chunks and reduces the memory usage by at least 50.7%; 3) reduces the total I/Os and average response time by up to 17.3% and 2.9% respectively.

VI. Acknowledgements

We thank anonymous reviewers for their insightful comments. This work is partially sponsored by the National 973 Program of China No.2015CB352403), the National 863 Program of China (No. 2015AA015302), the National Natural Science Foundation of China (NSFC) (No. 61332001, No. 61303012, No. 61572323 and No. 61628208), the Scientific Research Foundation for the Returned Overseas Chinese Scholars, and the CCF-Tencent Open Fund.

References

[1] D. Bhagwat et al. Extreme binning: Scalable, parallel deduplication for chunk-based file backup. In *Proc. of the IEEE MASCOTS'09*, 2009.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[3] W. J. Bolosky et al. Single instance storage in windows 2000. In *Proc. of the USENIX Windows Systems Symposium'00*, 2000.

[4] J. Bonwick. ZFS deduplication, 2009. *URL http://blogs.sun.com/bonwick/entry/zfs dedup*.

TABLE IV: Improvements of Zero-Chunk on memory consumption compared to Sparse Indexing

Trace	Homes-21					Web					Web Research				
Sampling Rate	1/64	1/128	1/256	1/512	1/1024	1/64	1/128	1/256	1/512	1/1024	1/64	1/128	1/256	1/512	1/1024
Improvements	56.98%	62.08%	73.33%	75.72%	79.10%	65.81%	67.98%	75.87%	73.57%	81.34%	54.29%	50.67%	64.21%	70.91%	72.35%

TABLE V: Improvements of Zero-Chunk on total I/O numbers compared to FIFO, LRU, LFU and Sparse Indexing

Trace	Homes-21					Web					Web Research				
Sampling Rate	1/64	1/128	1/256	1/512	1/1024	1/64	1/128	1/256	1/512	1/1024	1/64	1/128	1/256	1/512	1/1024
FIFO	63.22%	59.44%	59.86%	61.87%	59.95%	42.14%	41.64%	41.01%	42.96%	42.69%	29.09%	28.37%	31.39%	28.64%	33.24%
LRU	64.10%	60.24%	60.23%	62.20%	60.62%	40.38%	39.28%	39.80%	42.08%	41.84%	22.22%	15.02%	21.32%	21.21%	23.24%
LFU	63.74%	59.92%	60.08%	61.92%	60.07%	39.22%	38.85%	39.25%	41.57%	41.43%	17.54%	9.59%	13.95%	10.72%	17.02%
Sparse Indexing	17.34%	8.89%	10.68%	10.28%	5.80%	3.26%	1.98%	3.37%	2.49%	1.21%	4.68%	1.54%	1.35%	-0.77%	0.32%

[5] J. Bucy et al. The DiskSim simulation environment version 4.0 reference manual. Technical Report CMU-PDL-08-101, Carnegie Mellon University, 2008.

[6] Z. Chen et al. Ordermergededup: Efficient, failure-consistent deduplication on flash. In *Proc. of the USENIX FAST'16*, 2016.

[7] B. K. Debnath et al. ChunkStash: Speeding up inline storage deduplication using flash memory. In *Proc. of the USENIX ATC'10*, 2010.

[8] EMC Corporation. Executive summary - data growth, business opportunities, and the IT imperatives. http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm, 2014.

[9] G. Forman et al. Finding similar files in large document repositories. In *Proc. of the ACM SIGKDD'05*, pages 394–400. ACM, 2005.

[10] M. Fu et al. Design tradeoffs for data deduplication performance in backup workloads. In *Proc. of the USENIX FAST'15*, 2015.

[11] D. Harnik et al. Estimation of deduplication ratios in large data sets. In *Proc. of the IEEE MSST'12*, 2012.

[12] J. Kim et al. Deduplication in SSDs: model and quantitative analysis. In *Proc. of the IEEE MSST'12*, 2012.

[13] R. Koller et al. I/O deduplication: Utilizing content similarity to improve I/O performance. *ACM Transactions on Storage*, 6(3):13, 2010.

[14] M. Lillibridge et al. Sparse Indexing: Large scale, inline deduplication using sampling and locality. In *Proc. of the USENIX FAST'09*, 2009.

[15] B. Mao et al. POD: Performance oriented I/O deduplication for primary storage systems in the cloud. In *Proc. of the IEEE IPDPS'14*, 2014.

[16] D. Meister et al. dedupv1: Improving deduplication throughput using solid state drives (ssd). In *Proc. of the IEEE MSST'10*, 2010.

[17] D. T. Meyer et al. A study of practical deduplication. *ACM Transactions on Storage*, 7(4):14, 2012.

[18] K. Miller et al. XLH: More effective memory deduplication scanners through cross-layer hints. In *Proc. of the USENIX ATC'13*, 2013.

[19] A. Muthitacharoen et al. A low-bandwidth network file system. 35(5):174–187, 2001.

[20] W. C. Preston. Risk of hash collisions in data deduplication. http://www.backupcentral.com/mr-backup-blog-mainmenu-47/13-mr-backup-blog/145-de-dupe-hash-collisions.html, 2007.

[21] S. Quinlan et al. Venti: A new approach to archival storage. In *Proc. of the USENIX FAST'02*, 2002.

[22] M. Rabin. *Fingerprinting by random polynomials*. Aiken Computation Laboratory, 1981.

[23] R. Rivest. The MD5 message-digest algorithm. 1992.

[24] S.Mandal et al. Using hints to improve inline block-layer deduplication. In *Proc. of the USENIX FAST'16*, 2016.

[25] K. Srinivasan et al. iDedup: latency-aware, inline data deduplication for primary storage. In *Proc. of the USENIX FAST'12*, 2012.

[26] Y. Tian et al. Last-level cache deduplication. In *Proc. of the ACM ICS'14*, 2014.

[27] A. Verma et al. SRCMap: Energy proportional storage using dynamic consolidation. In *Proc. of the USENIX FAST'10*, 2010.

[28] W. Xia et al. Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput. In *Proc. of the USENIX ATC'11*, 2011.

[29] F. Xie et al. Estimating duplication by content-based sampling. In *Proc. of the USENIX ATC'13*, 2013.

[30] B. Zhu et al. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proc. of the USENIX FAST'08*, 2008.