2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th Intl. Conference on Data Science and Systems

PM-RAD: An Efficient Restore Algorithm in Deduplication by Pattern Matching

Guangping Xu*†‡, Yi Zhang*†‡, Sheng Lin*†‡, Kai Shi*†‡, Quan Yu §and Chi Wan Sung¶
*School of Computer Science and Engineering, Tianjin University of Technology, Tianjin, China
†Key Laboratory of Computer Vision and System, Ministry of Education, Tianjin, China
‡Tianjin Key Laboratory of Intelligence Computing and Novel Software Technology, Tianjin, China
Email:{xugp,zhangyi,linsheng,shikai}@email.tjut.edu.cn
§School of Information Engineering, Wuhan University of Technology, Wuhan, China
Email:yuquan@whut.edu.cn
¶Department of Electronic Engineering, City University of Hong Kong, Hong Kong, China
Email:albert.sung@cityu.edu.hk

***Abstract*—Deduplication is one of the most effective and efficient techniques to save memory space. It is widely used in data centers and cloud storage systems. After duplicated chunks are identified and removed, some logically consecutive chunks are physically scattered in different containers, which results in the serious fragmentation problem. The fragmentation problem inevitably leads the restore performance degraded severely. In this paper, we propose an efficient recovery algorithm by using pattern matching to boost the restore performance, which is called PM-RAD. It tries to reduce the number of contain reads by finding read patterns within a looking forwarding window. It also can merge scattered chunks and reads at once; thus it reduces the disk access times. Moreover, we optimize the proposed algorithm in two aspects, the separating caches and the cyclic pattern matching, to reduce disk accesses. During the pattern matching, we split cache into the metadata cache responsible for fingerprints and the data cache for storing chunks. The cyclic pattern matching ensures to find much longer patterns in a continuous sliding window. We implement the proposed algorithm and evaluate it by experiment with various data sets. Experimental results show that our algorithm is superior to the state-of-the-art work in terms of the restore performance.**

I. Introduction

Nowadays we are facing with the explosive growth of the digital data amount in big data era. It has become one of the most challenging and important tasks in mass storage systems to manage storage cost-effectively. Storage systems like data centers and cloud systems are unitizing data deduplication technologies to reduce redundant data and thus increase storage efficiency and reduce storage costs [1], [2], [3], [4], [5]. According to the references [6], [7], in a typical deduplication procedure, a *data stream* is segmented into *chunks* (also called blocks) and a cryptographic hash (e.g. MD5, SHA-1, SHA-256) is calculated as the *fingerprint* of a data chunk. The system checks whether a data chunk is redundant by comparing fingerprints instead of whole chunks. A *fingerprint index* maps fingerprints of the stored chunks to their physical addresses. To keep the spatial locality of stored data, a number of chunks are stored into a fixed-size storage unit, called a *container*, the basic unit of reads and writes. Those duplicate chunks are only stored their references to existing chunks in previous containers and those unique chunks are written into new containers. At the same time, the *recipe* records the references to the corresponding containers of incoming chunks. Thus the whole data are dispersed in containers physically while kept continuous in recipes logically, which could result in the serious *fragmentation problem*. It results in a great challenge to efficient restore [8], [9], [10], [11].

In this paper, we focus on speeding the restore process after data deduplication. During a restore, it needs to read data chunks from containers by using the recipe and fingerprint index; however, the fragmentation seriously constraints the restore performance due to reading the scattered chunks from different containers [8]. That is, the restore needs to read more containers randomly and then the restore performance becomes more poor due to the penalty of random disk seeks. Due to read some unnecessary chunks in containers, it wastes some disk bandwidth and degrades restore performance [4].

For deduplication-based backup storage, the restore speed for the most recent backup can drop by orders of magnitude over the lifetime of a backup system due to the chunk fragmentation problem. In order to solve this problem, most of existing solutions are to rewrite a small amount of fragmented duplicated chunks, such as [10], [13]. It trades deduplication ratio for restore performance. Some work tends to recycle invalid data blocks by identifying and merging valid scattered blocks of data into new containers. For example, the recent work proposed a cost-efficient rewriting scheme to improve restore performance [15]. In addition, cache replacement mechanism is a way to improve recovery performance. However, it is undeniable that the traditional cache replacement mechanism has been the ultimate, and the read performance has basically been unable to improve the efficiency of recovery.

From all the state-of-the-art work [12], [15], [16], we get the following **observations**: *a*) some rewritten chunks ensure some extents of data access locality; *b*) and caching mechanism should play an important role during the restore. Therefore, an efficient restore scheme should comprehensively consider how to digging effective I/O patterns and managing caches. Therefore, inspired by the state-of-the-art work on

978-1-5386-6614-2/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2018.00078

exploitation of data access locality and similarity and caching mechanism, the paper proposes a new recovery algorithm by using the pattern matching method, called PM-RAD. The key idea is to examine the placement of chunks and read distinct data chunks as continuously as possible, which result in reducing container reads as many as possible; moreover, cache prefetch mechanism should be reinforced.

In summary, the paper makes the following **contributions**.

- We propose a framework of the restore algorithm by maximum pattern matching. It tends to select containers with more referenced chunks for the backup, reducing the waste of disk accesses caused by redundant and unreferenced chunks in the restore. It can access less containers according to the matched patterns between the recovery list and the stored recipes by looking forward a window during the restore.
- We optimize the proposed algorithm by using the separated caches and cyclic pattern matching to speed the restore performance.
- we implement our algorithm in the deduplication system and evaluate the performance via various real-world backup datasets. Compared with the state-of-the-art schemes, experimental results demonstrate that our algorithm obtains better restore performance while keeping the desired deduplication ratio.

The rest of the paper is structured as follows. In section II, we give the background knowledge about data fragmentation and present the observations that motivate our work. In section III, we discuss related work. In section IV, We propose the recovery algorithm and show the experimental results in section V. In the final section, we conclude the paper.

II. Backgrounds and Motivation

To clearly describe our algorithm, this paper takes the same terms as the literatures [6], [7]. The typical restore process mainly have the following steps.

1) *the recovery list* is read from recipes. The fingerprints in the list are read and issued one by one.
2) For each fingerprint, the corresponding chunk is to read from a container into write buffer by lookuping fingerprint index.
3) Data chunks in write buffer are stored into disk sequentially.

We know that the data fragmentation degrades the read performance by increasing the number of container reads. Unfortunately, if the caching mechanism is unreasonable, the number of container reads would increase greatly, and the recovery performance would be unbearable for users.

Figure 1 illustrates the fragmentation process through two consecutive backups. The first backup has 14 data chunks, each distinct chunk marked with a distinct letter. These chunks including three duplicate ones are stored in 4 containers. Multiple data blocks in a backup stream may point to the same physical chunk. After the first backup, these chunks with 9 duplicate ones in the second backup are placed into 6

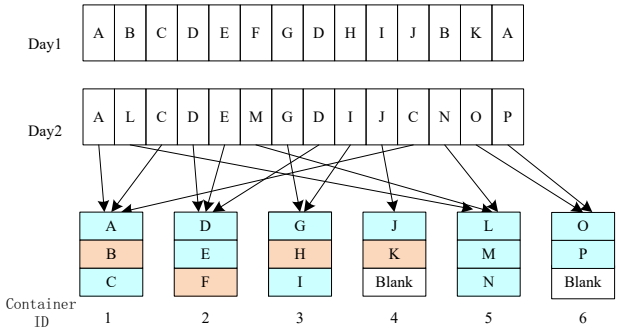

Fig. 1. The illustrative example of two consecutive backups in two days shows the chunk placement scattered in different contains, which is called the fragmentation problem.

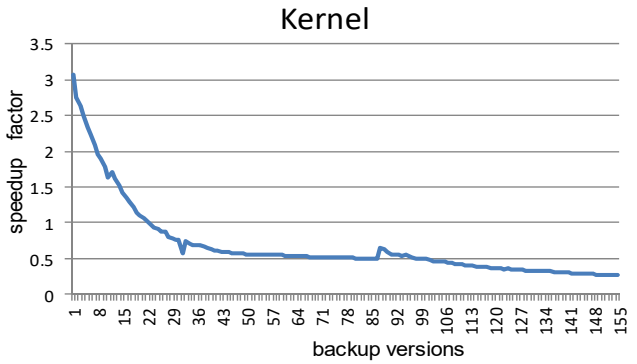

Fig. 2. The speed factor shows that the restore performance degrades with the increased backup versions. It leads to access more containers to restore the more recent backup version.

containers. It can be observed that the more backups and the more fragmented data chunks becomes. With the condition that the size of cache is three container sizes and the replacement policy is LRU, the first backup would need to read the containers 5 times but the second backup would need 9 times. So it can be observed that the more backups, the worse restore performance.

As a metric, the speed factor indicates the extent of data fragmentation and is equal to the amount of total data divided by the number of containers read. That is, it is such a metric that 1 is divided by the number of containers each 1 MB data needs to read [6]. Figure 2 shows the speed factor of the 155 backup versions. Higher speed factor means that less containers are needed for the restored data per 1 MB data, thus indicating better restore performance. From the figure, the speed factor decreases as the increasing of the backup versions: the more backups are done, the more fragmented chunks are placed. In particular, the fragmentation problem of the latest version is the worst; that is, it will take the longest time to recover the version. A few exceptions in the Linux kernel data sets are the major revision updates, which have more new data stored consecutively. So we focus on relieving the performance degradation of the restore over time caused

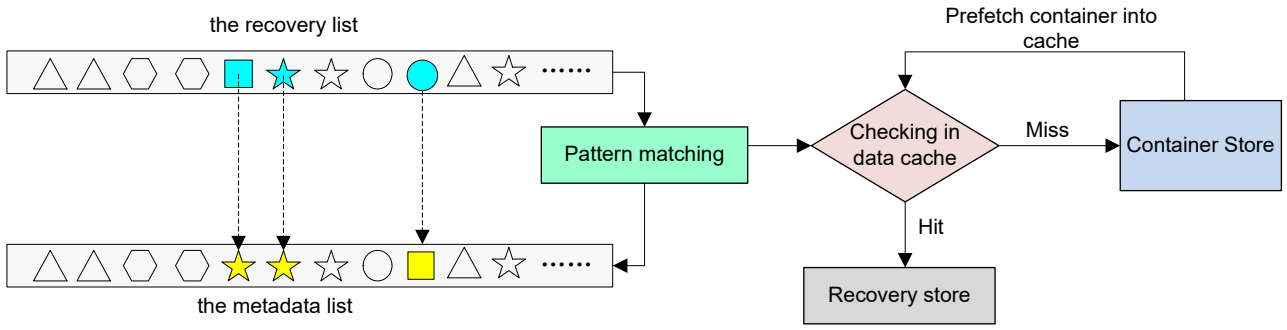

Fig. 3. The diagram of the proposed restore algorithm by using pattern matching. The access patterns are matched between the recovery recipe and the stored fingerprint list.

by the fragmentation problem.

For data cache, the design of an appropriate replacement policy is critical to improve restore performance in deduplication systems. Assuming the replacement policy is FIFO, it is not a good policy because the utilization of containers is not taken into account. Moreover, the LRU policy requires the recent used container retains in data cache; however, the policy does not consider the future access. Since the sequence of read chunks during the restore is the same as backup recipes, we can determine to cache a container or not in advance.

Therefore, the motivation of our work tries to explore the access pattern from the placement of chunks in containers and take full advantage of data cache; thus our goal is to access containers as few as possible.

III. Related Work

After identifying duplicate data and storing the nonduplicate data, logically consecutive chunks are physically scattered in different containers after deduplication, which results in the serious fragmentation problem. The fragmentation significantly reduces the restore performance due to reading the scattered chunks from different containers. It is desirable to efficiently restore data which requires read the fragmented chunks efficiently. The state-of-the-art data restore work often rewrites some duplicate but fragmented chunks to alleviate the degradation of restore performance. So it involves the tradeoff between deduplication ratio and restore performance.

At the earlier time, iDedup [4] adopted a simple scheme in the primary storage, which deduplicates a sequence of chunks whose physical addresses are also sequential exceeding a minimum length threshold. Similarly, Nam et al. [8], [9] proposed to selectively eliminate sequential and duplicate chunks with a quantitative metric called chunk fragmentation level (CFL). Context-based rewriting (CBR) [13] and Capping [10] algorithms determine the fragmented chunks in the write buffer using their specific fragmentation metrics, and then selectively write the fragmented chunks to improve restore speed. In particular, capping uses a forward assembly technique to efficiently cache chunks by exploiting the perfect knowledge of future chunk accesses available when restoring the already known backups. Moreover, RevDedup [14] eliminates duplicates from the previous backups while conventional deduplication eliminates duplicates from the new backups. It transfers fragmentation to old backups and tries to keep the latest backup as sequential as possible. History-aware rewriting tries to rewrite less data to achieve better restore performance by accurately classifying fragmentation [11], [12]. The work of this paper is complemental to all these rewrite schemes, which can be integrated into a whole deduplication system. Specifically, after rewriting, our algorithm could find more better patterns to read chunks during the restore, which can yield better restore performance. In the implementation of our algorithm, we analyze the effect of the capping rewrite scheme.

While the rewriting algorithm determines the chunk placement, an efficient restore algorithm leverages the placement to gain better restore performance with a limited memory footprint. There have been three restore algorithms: the basic LRU cache, the forward assembly area (ASM) [10], and the optimal cache (OPT) [11]. In all of them, a container serves as the prefetching unit during a restore to leverage locality. Their major difference is that while LRU and OPT use container-level replacement, ASM uses chunk-level replacement. We will compare these algorithm performance with our proposed algorithm under different placements in section V.

IV. Proposed Algorithm

In this section, we first describe the methodology of our proposed restore algorithm based on pattern matching, called PM-RAD; and then after giving the algorithm framework, we describe how to optimize it in more detail.

A. Methodology

In this subsection, we present the whole restore process, as shown in Figure 3. At first, it finds the maximum common sequence named a pattern by matching the recovery list and the storage list. And then it reads data chunks in a pattern from a container at once and prefetching the container into data cache. In this paper, the container is still a prefetching unit except that the number of chunks is less than half of a container.

The cache acts as the acceleration of data lookups exploring the locality of data streams. That is, we prefetch the useful container into cache to save the next container read from disk. We separate the whole cache into the meta-data cache (C_m) and the data cache (C_d). By the meta-data cache, it caches the metadata of containers to speed up matching pattern. Specifically, the size of metadata of a container is relative small; and thus when reading metadata of a container to form a matching list, the access time can be ignored. For the data cache, only those containers which will be accessed often identified based on patterns are allowed to enter data cache. Moreover, various rewriting methods potentially help to extend the length of patterns.

In the proposed algorithm, the most crucial step is to match the recovery list which read from the backup recipes and the storage list which is read from the metadata of a container. The recovery list records the fingerprints of data chunks of the stored files and their corresponding container id. The storage list records data chunks stored in a container. Both lists can be read efficiently and their memory footprint is small. Due to the limit size of cache, it is impossible to match all recovery recipes with the storage lists; we need to slice the recipe list into segments. At each matching iteration, only a certain amount of fingerprint is read as a segment for matching and the length of a segment is a critical factor for the performance of reading. As the time complexity, each pattern matching can be solved with $O(n)$ time total by the classical Longest Common Subsequences (LCS) algorithm [18].

Algorithm 1 Basic algorithm framework by using maximum pattern matching

Input: The list of fingerprints to recover chunks, called the recovery list, R; The size of the looking forward window, m; The length of wildcards in the matching process, x.

Output: The set of subseqences of matched fingerprints, denoted as P;

1: Read m fingerprints from R as a segment S;
2: Determine the container which the start item in S belong to;
3: Read the metadata of the container, its fingerprint list called T;
4: Find the longest common sequence between S and T, denoted as lcs;
5: Get the access patterns from lcs based on the wild matching size x and put them into P;
6: Remove the fingerprints which appear in patterns from S;
7: Repeat the steps above until S becomes empty;

B. Maximum pattern matching

The framework of the pattern matching process in PM-RAD is described in Algorithm 1. During a pattern matching, the algorithm slides forward a part of the recovery list every time, called a slide window, denoted as m. In the algorithm, the number of wildcards is configured by the parameter x; the number of fingerprints in a slide window is configured by the parameter m; these parameters are initialized at first. When the pattern matching, the non-adjacent fingerprints are not beyond the interval length x in storage list are considered continuous in a pattern. That is, these x wildcards are to skip a certain adjacent fingerprints to generate a longer pattern from the storage list. Thus the sequent disk access by a longer pattern yields better read throughput. Considered the disk access mechanism, the seek time and read time greatly affect the restore performance. The sequent disk access guided by pattern is more efficient than the random separate disk access.

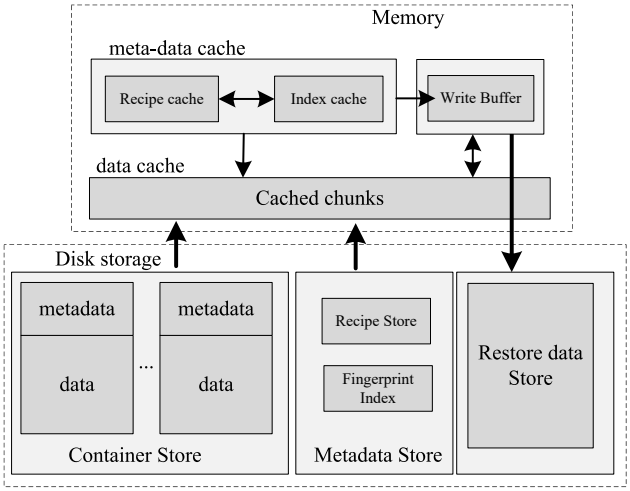

Fig. 4. The architecture of the implementation of the proposed algorithm.

Here consider the following example. Assume that a recovery list is $ALCDEGDMIJANO\ldots$; and by reading the metadata of the container in which the A is located, the metadata list is $ABCDEFGHIJKNOP\ldots$. Then the matched pattern is $ACDEG$ if $m = 8$ and $x = 0$; furthermore, if $x = 1$ then a longer pattern is obtained $A*CDE*G$ where $*$ represents a wildcard.

C. Algorithm optimization

1) Separated caches: The whole storage architecture include caches and persistent disk is shown in Figure 4. As mentioned in section III, a full cache splits into two parts, namely, the metadata cache and the data cache. The metadata of a container records all fingerprints and their referred address in the container. Such the metadata of a container can be cached easily while data chunks of a container consume much more data cache. In the deduplication system, an backup data stream place data chunks in different containers. By reading a recipe from the recipe store and lookuping the fingerprint index, all locations of chunks can be determined one by one; then some containers need to store in data cache. During the pattern matching process, chunks which has been recovered in a sliding window are stored into write buffer temporarily. After all chunks in a sliding window are in write buffer, then write them into disk in a batch. Therefore, the data cache management as the important component of our algorithm is

critical to boost restore performance. It largely depends on the access pattern of containers. In the next subsection, we present a novel pattern matching method to find better patterns.

Algorithm 2 PM-RAD: the optimized algorithm by using separated caches and cyclic pattern matching

Input: The list of fingerprints to recover chunks, called the recovery list, R; The size of the looking forward window, m; The length of wildcards in the matching process, x;

Output: The set of sequences of matched fingerprints, denoted as P;

```
1: Allocate data cache C_d for data chunks;
2: Allocate meta-data cache C_m for fingerprints.
3: Read a segment S_1 from R;
4: while S_1 is not empty do
5:     if S_2 == ∅ then
6:         Read the next segment S_2 from R;
7:     end if
8:     Remove items whose chunks appeared in C_d from S_1;
9:     Write those data chunks of S_1 in C_d into buffer;
10:    if the current size of S_1 ≤ m/2 then
11:        S_1 += S_2
12:        S_2 = ∅
13:    end if
14:    if the first item of S1 is no in C_m then
15:        Read the metadata of its container T into C_m
16:    end if
17:    Find the longest common sequence between S_1 and
       T, denoted as lcs;
18:    Get the access patterns from lcs based on the wild
       matching size x and put them into P;
19:    Read chunks of the container with which the longest
       pattern associates.
20: end while
```

2) Cyclic pattern matching: During the pattern matching in Algorithm 1, we observe it has the following weakness: S_1 becomes shorter after each matching iteration, which results the further found pattern becomes shorter. So it affects negatively the restore performance. In order to avoid S_1 becoming too short, the other list called S_2 is read from the recovery list ahead; when the current length of S_1 is less than the half of its original length, then S_1 merges S_2 for the next matching and fill S_2 again. So the list S_2 compensates S_1 in a cyclic way to keep S_1 enough long, which ensures to find better patterns.

Besides of the benefit to find better patterns, the list S_2 provides some hints to evict some container from data cache. We can take the length of S_2 as a sliding window in the nearest future; thus from S_2 we know which container cached will be no useful which can be evicted. We try to cache those containers which will be referenced by fingerprints in S_2. Note that the size of the sliding window S_2 is determined by the data cache size to a large extent; moreover, when more cached containers are no useful, then adjust S_2 longer.

V. Experimental Evaluation

we implemented and evaluated the proposed algorithm with different settings driven by various data sets. In this section, we first describe the experimental setup with various data sets; then we present the experiment results with different setting, segment sizes and cache sizes; we compare the restore performance of the proposed algorithm with other algorithms. Finally, we show the positive effects of rewrite to restore performance.

A. Experimental settings

We implemented our proposed algorithm in a well-known open source system, called Destor [6]. We released the source code of PM-RAD at the github site [17].

The following experiments were run on Ubuntu $14.04.5$ LTS operating system. The experimental hardware includes a quad-core CPU running at 2.4 GHz, with 32 GB RAM and two 1 TB $7,200$ rpm hard disks. We employed four data sets, namely, Kernel, VM, FSLhomes and Macos to drive our experiments. The details of these data sets are briefly described as follows.

- Kernel is downloaded from the web [19]. Here 155 versions of unpacked Linux kernel source code are chosen for our experiments. The data set shows high locality between adjacent versions.
- Vm refers to pre-made VM disk images from VMware's Virtual Appliance Marketplace, which is often used to explore the effectiveness of deduplication on virtual machine disk images.
- FSLHomes is published by the File system and Storage Lab (FSL) at Stony Brook University[20]. It contains snapshots of students' home directories. The files consist of source code, binaries, office documents, and virtual machine images.
- MacOS is also the data set published by FSL. It includes snapshots on a Mac OS X Snow Leopard server supplying various services for hundreds of users [21].

All of these data sets provide enough data to drive experiments and then facilitate us to evaluate the proposed algorithm. In this paper, we measure the restore time in the experiments, i.e., the lower the measured value, the better the restore performance; moreover, we also examine the deduplication ratio which is the ratio of the removed duplicate chunks to that of all backup chunks.

B. Sliding window size

Recall that the sliding window size is an important parameter which directly affect the lengths of the matched patterns. In the experimental implementation, we slide the recovery list forward by a segment algorithm. Here the sliding window size is called a segment size. The unit of each segment is a fingerprint. The segment size is set from 256 to 8192 for VM and from 64 to 2048 for Kernel, FSLhomes and User006 in FSLhomes. The experimental results are shown in Figure 5. We observe that all these curves have the similar shapes: the restore time shows a trend from decline to rise with the increasing segment size; the restore achieves the

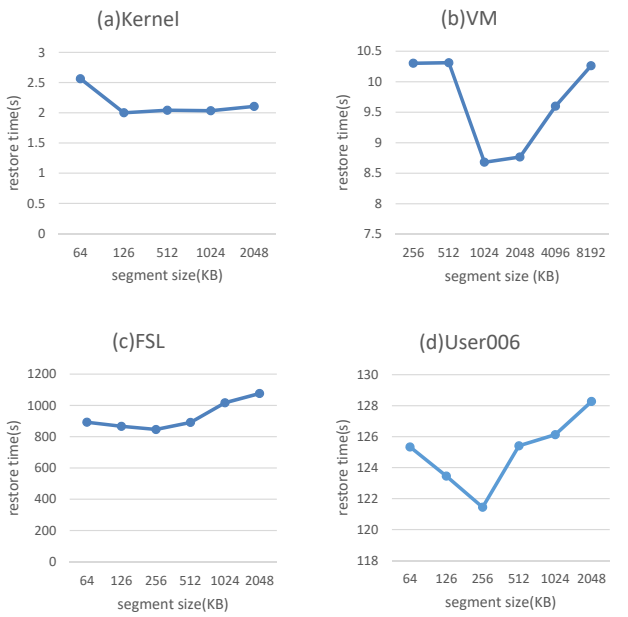

Fig. 5. The restore time as segment size varies. All of these curves have the similar shapes: the restore time first deceases and then increases when increasing the segment size.

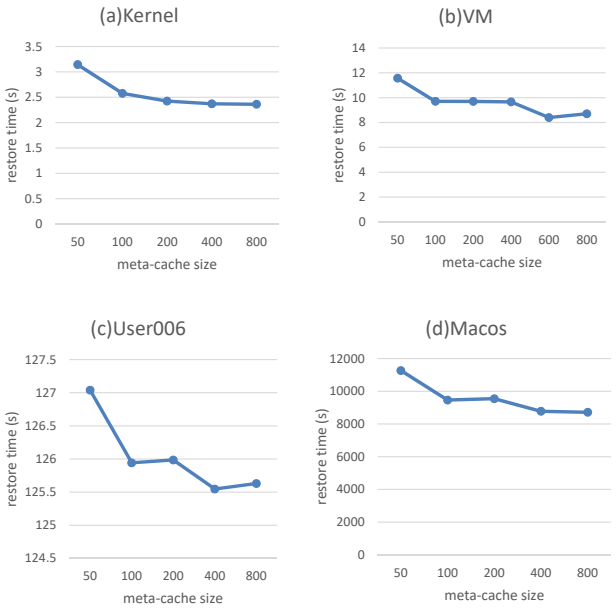

Fig. 6. The restore time as the size of metadata cache varies from 50 to 800, which shows the larger cache is helpful to reduce the restore time.

best performance when the sliding window size is 128, 1024, 256 and 256 for the four datasets, respectively. We have the following observations.

- *First*, these results confirm and validate that the sliding window size affects the performance of the pattern algorithm greatly. Note that in the following experiments, the sliding window sizes are configured with these values. It is desired to adaptively set the parameter during the restore.
- *Second*, the increasing segment size leads to generate more longer patterns reduce; so the number of disk accesses are reduced. However, when the segment size is too large, the data cache becomes the major performance constraint. Some containers are prefetched too early into cache. That is why these curves drop at first and then rise.
- *Third*, due to the workload differences of the datasets, it leads to different suitable segmentation sizes. For example, the experiments show that the User006 data set, a subset of FSLhomes, spend less than 85% on the average overhead of each backup comparing with FSLhomes. This is because the workloads of a lot of users in FSLhomes are far more mixed than a single user, User006. So the data locality is poor in FSLhomes and the restore time is 6 times as much as the recovery time of User006.

C. Meta-data cache sizes

The method based on pattern matching is to optimize container access. It is equivalent to exploit the access locality of containers. As well known, the larger cache usually results in a high cache hit ratio. The metadata cache is configured too small to prefetch enough metadata into cache, resulting the decrease of the rate hitting in the metadata cache and the increase of each recovery time. So the larger the metadata cache is, the more time is saved. As the cache capacity increase, the recovery is expected to yield better performance because there is enough space to buffer more metadata and the looking forward more fingerprints in S_1 and S_2 (see Algorithm 2).

In Figure 6, we observe that the read performance with the variation of metadata cache size. Here considering the memory consumption, we choose the total meta-data size of 100 containers as the meta-cache size. The results show that the curve falls gradually as the meta-data cache size increases and tends to be more stable when the cache size increases up to 100 at x-axis. When the cache can hold the metadata of 100 containers, it has a lower replacement rate, saving a lot of time in reading the same container. Even if the cache size is set to 800, the restore performance gains are limited, 9.1%, 15.2%, 0.2%, 8.2% for Kernel, Vm, User006, Macos, respectively. So the data cache preserves the containers that have been read; in case that those containers are read again in the near time, it is helpful to accelerate the access performance.

D. Comparison with other algorithms

With the optimal segment sizes as described in Figure 5, Figure 7 shows the comparison of three replace cache policies and pattern matching methods over the Kernel and FSLHomes data sets without rewriting. For kernel data set, the data cache size of ASM, LRU and OPT is set 100 containers while for FSLHomes it is set 400 containers. In the experiments with

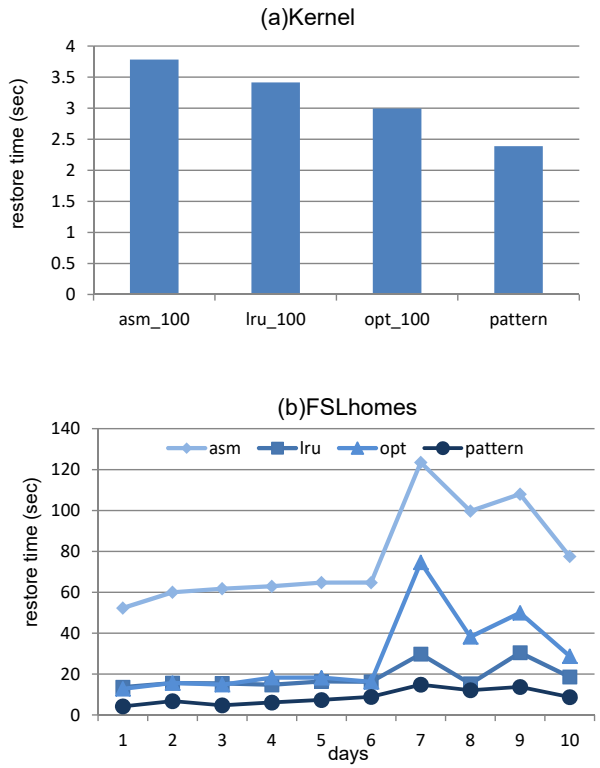

Fig. 7. The restore performance comparison of our algorithm with different restore algorithms including LRU, ASM and OPT, which are mentioned in Section III.

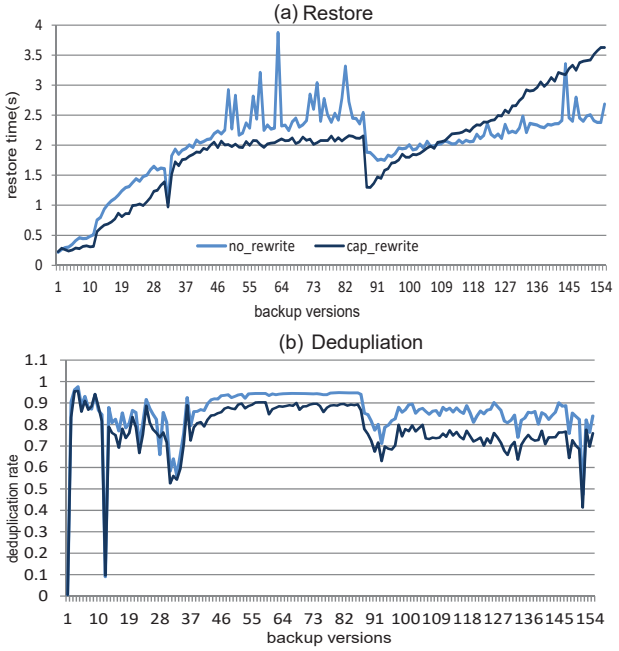

Fig. 8. For kernel data set, the rewrite effects to the restore by using the capping rewrite scheme. During the evolution, the restore time and deduplication rate are depicted. It can observed that, the restore with rewrite usually gets better restore performance than without rewrite at the cost of certain deduplication rate.

both the datasets, PM-RAD avoided the data cache at first and gradually allocated cache for containers. The algorithm has known the distribution of chunk in container, as follows, selects the continuous reading or the single reading according to the distance between two chunks in one segments. The reasonable pattern read scheme greatly saves the restore time. By the experimental result as shown in Figure 7, we observer that our proposed pattern matching method is superior to the state-of-the-art methods. For the Kernel dataset, the restore performance of PM-RAD is 25.68%, 42.9%, 58.44% higher than the performance by OPT, LRU and ASM, respectively. For FSLHomes, the restore performance of PM-RAD is speedup up to 2 times, 3 times, 8 times, compared with OPT, LRU and ASM, respectively. These experimental results show that PM-RAD is superior to the state-of-the-art work in terms of the restore performance.

E. Rewrite effects

Rewrite refers to at the cost of a certain amount of storage space at the time of backup to reduce the fragmentation of backup data, thus speeding up the recovery performance and inhibiting the recovery performance degradation. Here with the capping rewriting technology that rewrites a little of fragmented data block, the pattern matching method extended more continuous reading, so as to improve the reading performance. In comparison with figure a that depicts the trend of each backup, Figure 8(a) describes the restore performance of 155 backups. And the rewriting technology assuredly contributes our algorithm to the improvement of performance. We observe that with the increase of the number of backup versions, more and more duplicate chunks are rewritten into containers. The restore performance is improved before the 110-th backup; however, the restore performance tends to the restore performance of the algorithm without rewriting. The reason may be the fragmentation problem becomes too server after too times of backups are done. It illustrates that it still is a challenge to design an effective rewrite algorithm for long-term backups.

In Figure 8(b), it depicts the deduplication with cap rewriting and the one without rewriting technology. It can be observed that the increase of rewriting only reduces the rate of less than 10%. It can get more read performance at the expense of a small amount of deduplication. To speedup data recovery, after tradeoff recovery performance and deduplication, we can choose to rewrite partially fragmented data to improve the performance of data recovery.

VI. Conclusion

In this paper, we proposed an efficient restore algorithm by finding maximum matching patterns between the recovery list and storage list to solve the fragmentation problem in deduplication systems. When restoring data, we compute the longest common sequence by pattern matching. It reads all chunks of a pattern from a container at once. We further optimize the proposed algorithm from both ways. One way is to keep patterns enough long and the other way is to keep useful containers cached. So it reduces the number of container reads as much as possible. Our algorithm prevent the degradation of restore performance as the increasing number of backups. We implemented the algorithm and conducted elaborated experiments to evaluate the restore performance. By the experimental comparison with the state-of-the-art work, our proposed algorithm significantly improves the performance of data recovery. Since the rewrite mechanism is still an open problem, the restore pattern embedded into rewriting may be worth investigating in the future.

Acknowledgment

We are grateful to Dr. Min Fu who gave some helps on the experimental platform Destor and prof. Gang Wang in NBJL who gave some valuable feedback on the work. The work was partly supported by NSFC projects(No.61201234, 61202381 and 61170301) and NSF projects of Tianjin (No. 17JCYBJC15600, 15JCQNJC00500);

References

[1] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage," in Proc. FAST, 2002.
[2] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in Proc. FAST, 2008.
[3] C. Dubnicki et al., "Hydrastor: A scalable secondary storage," in Proc. FAST, 2009.
[4] K. Srinivasan, T. Bisson, G. R. Goodson, and K. Voruganti, "iDedup: latency-aware, inline data deduplication for primary storage," in Proc. FAST, 2012.
[5] A. El-Shimi, R. Kalach, and A. Kumar et al., "Primary data deduplication large-scale study and system design," in Proc. USENIX ATC, 2012.
[6] Min Fu et al., "Design Tradeoffs for Data Deduplication Performance in Backup Workloads," in Proc. FAST, 2015.
[7] 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, 2016.
[8] Y. Nam, G. Lu, N. Park, W. Xiao, and D. H. Du, "Chunk fragmentation level: An effective indicator for read performance degradation in deduplication storage," in Proc. HPCC. IEEE, 2011.
[9] Y. J. Nam, D. Park, and D. H. Du, "Assuring demanded read performance of data deduplication storage with backup datasets," in Proc. MASCOTS. IEEE, 2012.
[10] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in Proc. FAST, 2013.
[11] M. Fu et al., "Accelerating restore and garbage collection in deduplicationbased backup systems via exploiting historical information," in Proc. USENIX ATC, 2014.
[12] M. Fu et al., "Reducing fragmentation for in-line deduplication backup storage via exploiting backup history and cache knowledge," IEEE Trans. Parallel Distrib. Syst., vol. 27, no. 3, pp. 855-868, 2016.
[13] M. Kaczmarczyk, M. Barczynski, W. Kilian, and C. Dubnicki, "Reducing impact of data fragmentation caused by in-line deduplication," in Proc. SYSTOR. ACM, 2012.
[14] Y. Li, M. Xu, C. Ng, and P. P. Lee, "Efficient hybrid inline and out-of-line deduplication for backup storage," ACM Trans. Storage, vol. 10, no. 2, pp. 2-21, 2014.
[15] J. Wu, Y. Hua, P. Zuo and Y. Sun, "A Cost-efficient Rewriting Scheme to Improve Restore Performance in Deduplication Systems," in Proc. MSST. IEEE, 2017.
[16] Y. Zhou, Y. Deng, L. Yang, R. Yang, L. Si. LDFS: A Low Latency In-line Data Deduplication File System, IEEE Access, 2018.
[17] PM-RAD source code. https://github.com/xugp2008tj/PM-RAD.
[18] M. T. Goodrich and R. Tamassia. "Algorithm Design and Applications," Wiley, 2015.
[19] Linux kernel. http://www.kernel.org/.
[20] FSLhomes. http://tracer.filesystems.org/traces/fslhomes/.
[21] MacOS. http://tracer.filesystems.org/traces/macos/
[22] VM. http://www.thoughtpolice.co.uk/vmware/.