Improve Restore Speed in Deduplication Systems Using Segregated Cache

Wenjie Liu[†], Ping Huang[†‡], Tao Lu[‡], Xubin He[‡♭], Hua Wang[†*], and Ke Zhou[†]

[†]Wuhan National Lab for Optoelectronics, Huazhong University of Science and Technology, China
[‡]Department of Electrical and Computer Engineering,Virginia Commonwealth University, USA
[♭]Department of Computer and Information Sciences, Temple University, USA

***Abstract*—The chunk fragmentation problem inherently associated with deduplication systems significantly slows down the restore performance, as it causes the restore process to assemble chunks which are distributed in a large number of containers as a result of storage indirection. Existing solutions attempting to address the fragmentation problem either sacrifice deduplication efficiency or require additional memory resources. In this work, we propose a new restore cache scheme, which accelerates the restore process using the same amount of cache space as that of the traditional LRU restore cache. We leverage the recipe knowledge to recognize the containers which will soon be accessed for restoring a backup version and classify those containers into *bursty* containers which are differentiated from other *regular* containers. *Bursty* and *regular* containers are then put in two separate caches, respectively. *Bursty* containers, containing many chunks that will be needed for restore within a short period of time, are put in a smaller cache managed at the container granularity. On the contrary, *regular* containers are put in the other bigger cache managed at the chunk granularity, with chunks which will not be used dropped off at the time when the containers are brought in. In doing so, bursty containers have better chances to be quickly evicted from the restore cache, avoiding their unnecessarily occupying cache space for too long. Our evaluation results have demonstrated that our proposed cache scheme can improve restore speed factor by up to 3.05X and reduce the number of container reads by 67.3% on average, relative to a conventional LRU restore cache.**

1. Introduction

Data deduplication, a technique that identifies and removes duplicate content in storage systems, has become an essential feature of modern storage systems to reduce storage requirements [10], [19], [20], both in primary storage [2], [14], [15] and backup appliances [6], [7], [20]. Digital information has been generated at an accelerated rate during the past decades [1], [12]. According to the 2014 IDC forecast, the data that we create and copy is growing 40% a year into the next decade and by 2020 the yearly generated data will reach 4.4 zettabytes [17]. To own and efficiently manage such a huge volume of data imposes tremendous challenges to data owners. Data deduplication turns out to be effective in coping with the problem of owning such "big data". Typically, for backup datasets, a 10-30X of space savings can be achieved with data deduplication [8], [18]. We concern deduplication used in backup environments.

Data deduplication eliminates redundant content via employing mechanisms to recognize duplicate content and only storing a single copy of duplicate data. In deduplicated backup systems, an incoming backup data stream is first divided into individual chunks, using *fix-sized* [13] or *variable sized* [20] chunking algorithms. For each resultant chunk, a *fingerprint* is then calculated using a cryptographic function, e.g., MD5 or SHA1. Fingerprints are used as proxies to check chunk uniqueness. All fingerprints are stored in a so-called *index store*, which records all pairs of chunk fingerprints and their associated physical locations (e.g., containers [3], [4], [20]). At ingesting time, every incoming chunk fingerprint is checked against all the fingerprints (for exact deduplication) or part of the fingerprints (for near-exact or approximate deduplication) [4], [9] which have previously been stored in the *index store*. If there exists no same fingerprint, the chunk represented by the fingerprint is assumed to be a new and unique chunk, which should be ingested into the storage system. However, if a same fingerprint has been found, then the corresponding chunk is said to be duplicate and the system need not store the chunk. Instead, it simply remembers the chunk location. For each backup, the system creates a *recipe* [11] to remember all pairs of constituent fingerprints and their corresponding physical addresses (e.g., containers IDs).

As time goes on, chunks of later backup versions tend to be distributed in increasingly large space, causing the so-called "chunk fragmentation" problem. The chunk fragmentation problem results from duplicate chunks existing both in the same backup stream and across multiple backup streams. To restore a chunk-fragmented backup version, it may require to access a large number of containers, dramatically degrading restore performance. To reduce container reads, existing solutions choose to either slightly sacrifice deduplication efficiency at backup time by selectively rewriting duplicate chunks [3], [5], [8] or use additional memory resources [6], [8] at restore time. In this work, we suggest a new restore cache scheme which is able to reduce

*Corresponding author.

2375-0227/16 $31.00 © 2016 IEEE
DOI 10.1109/MASCOTS.2016.46

IEEE computer society

container reads, while obviating the shortcomings/overheads existing with previous solutions. We call the new restore cache *FS-CARE* as it is aware of chunk *F*ragmentation and *S*parseness in the containers. The key idea of *FS-CARE* is to leverage the fact that the recipe, which has full knowledge of chunk information for restore can be used to characterize the containers into either *bursty* or *regular* containers. *Bursty* containers are containers that contain a relatively large number of chunks which are needed in a short period of time window. Correspondingly, we partition the restore cache into two separate caches to accommodate the two types of containers, respectively, as different types of containers have distinct access behaviors. The hope is to quickly evict those containers which are accessed in a bursty manner from the restore cache to avoid keeping them in cache for too long, without missing usage of the chunks contained in the container. Evaluation results have shown that our proposed restore cache is effective in reducing container reads and improving speed factor.

The rest of the paper is structured as follows. In Section 2, we give the background knowledge about restore operations in deduplication systems and present the observations that motivate our work. We then elaborate on our restore cache design in Section 3, which is followed by evaluation results in Section 4. We discuss related work in Section 5 and conclude the paper in Section 6.

2. Background and Motivation

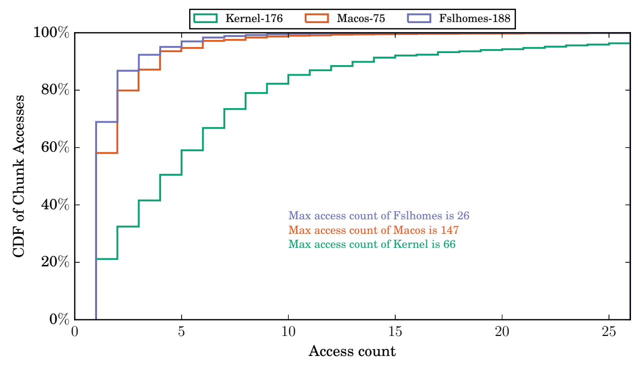


Figure 1: The figure shows the access count CDFs of three restores for the three workloads. The backup version restored to is denoted in the respective legends. About 10% and 50% containers were accessed more than 5 times for *Macos* and *Fslhomes*, *Kernel*, respectively.

The restore process in deduplication systems mainly consists of three steps. First, the recipe corresponding to the restore backup version is read. Typically, the recipe is read in an incremental fashion, meaning only a portion of the recipe is read in at a time. Second, the chunks referred to by the read recipe are fetched in a restore cache. Chunks are accessed in the granularity of container, i.e., the whole container to which a referred chunk belongs is brought into the restore cache to satisfy a chunk request. The other remaining chunks in the same container are also put in the restore cache in anticipation that when they are needed in the future, a read to the same container can be avoided. Finally, the chunks are then written out to construct the original backup. Chunks need to be written out sequentially to the restore destination, though memory buffer can be deployed to temporally hold chunks for reordering before they are written out sequentially as a batch [8]. The restore cache is commonly managed as an LRU cache to leverage chunk locality which results from the following observations. During backup, if a chunk is new, then it is likely that the following chunks are also new, and if a chunk is duplicate, then it is also very likely that neighboring chunks are also duplicate. In both cases, the neighboring chunks are clustered in the same container.

However, we argue that an LRU-managed container-based restore cache may not be effective due to the existence of chunk fragmentation. To verify that, we have conducted experiments to investigate the efficacy of the LRU-managed container-based restore cache from various aspects and found several interesting findings which serve as our motivations. Experimental setup details can be found in Section 4. First, a container may be read in multiple times to satisfy the chunks belonging to that container. Figure 1 shows the chunk access CDFs of three randomly selected restores for the three workloads. As can be seen from the figure, there are approximately 10% containers which are read in more than 5 times to perform a restore for *Macos* and *Fslhomes* workloads and the number becomes 50% for *Kernel* as it suffers from more severe fragmentation (see Table 1). Second, the container utilization (defined as the ratio of the number of chunks accessed to the container size for each container read) is low due to the frequent container swapping in and out. Figure 2 shows the container utilization of the last 1000 containers for a *Fslhomes* restore. As it is shown, a significant amount of containers have lower utilization. Actually, the average utilizations of the three restores in Figure 1 are 59.8% (for *Fslhomes*), 74.5% (for *Macos*), and 3.1% for *Kernel*, respectively. Third, due to fragmentation, it needs to access many containers to perform a restore. Figure 3 shows the number of container reads to restore every 5K chunks for different restores. Please note that a 4MB container contains about 1K chunks. However, as can be seen, the majority of restores need to access more than 100 containers (a total of 100K chunks) to restore 5K chunks, causing a container read amplification factor of 20. To summarize, our experiments have demonstrated that an LRU container based restore cache causes multiple accesses to the same containers, lower container utilization, and excessive unneeded chunks to brought in the cache. Our goal in this work is to reduce the amplification factor by leveraging the restore cache in a more efficient manner.

3. Restore Cache Design

In this section, we first qualitatively research the reasons why an LRU-managed container-based restore cache might not be effective and how it can be improved via studying

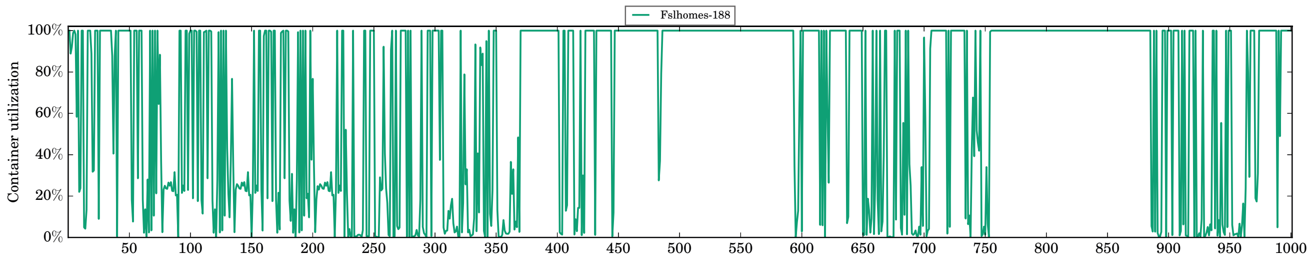


Figure 2: It shows the container utilization of the last 1000 containers for a restore of *Fslhomes*. The average container utilization is 59.8%. If a container has good locality, the container utilization is high. Similarly, if it has poor locality, the container utilization is low.

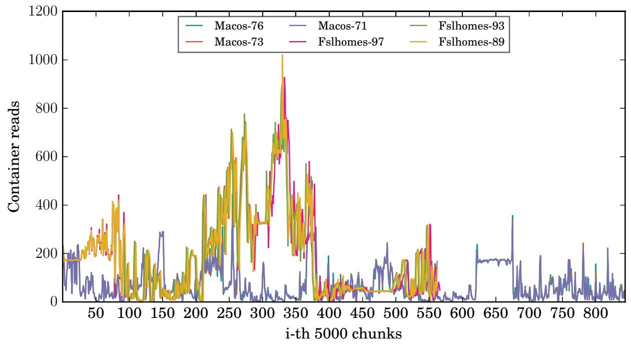
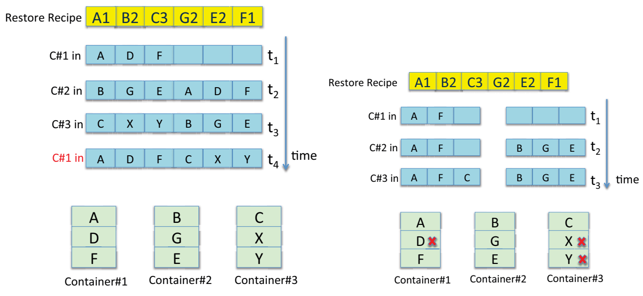


Figure 3: This figure shows the number of container reads to restore every 5K chunks for the three workloads. As it is shown, in the majority scenarios, it requires to read more than 100 containers, i.e., 100K chunks, in order to restore 5K chunks.

a simple restore cache example. Based on that, we move forward to discuss our cache design details, particularly the mechanisms that are deployed in the design, which improve the restore cache efficiency by reducing unnecessary container reads.

3.1. Excessive Container Reads

As it is shown in Figure 4a, we assume that the instant state of a deduplication system at some time is the same as what is depicted in the figure. On top of the figure is a portion of the recipe knowledge, in the middle is the restore cache status which evolves as time goes on, and in the bottom is the storage status of the underlying storage containers. Each recipe entry is denoted by a combination of a character and a number, with the character being the chunk content and the number denoting the container to which the chunk belongs. With an LRU-managed container based restore cache, as can be seen from the figure, it requires to read a total of four containers in order to finish restoring the chunks contained in the given recipe, i.e., at time t_1 Container $\#1$ is read to satisfy chunk A, at time t_2 Container $\#2$ is read to satisfy chunk B, at time t_3 Container $\#3$ is read to satisfy chunk C, and finally at time t_4 Container $\#1$ is read again to satisfy chunk F. As we will see shortly, the last container read could actually be obviated if we had managed to maintain chunk F in the cache after it was brought into the cache for the first time (i.e, at time t_1). This simple illustrative example has demonstrated the potential limitations of an LRU restore in incurring unnecessary container reads.



(a) LRU restore cache (b) FS-CARE restore cache

Figure 4: Figure 4a shows a conventional LRU restore cache which incurs four container reads. Figure 4b illustrates that our proposed segregated cache can reduce one container read for the same restore.

3.2. FS-CARE Cache Design

To reduce unnecessary container reads shown in Figure 4a, we suggest integrating two mechanisms to the baseline LRU restore cache. First, we leverage the recipe knowledge to characterize the containers that will be needed in the given recipe window into *bursty* and *regular* containers. Basically, bursty containers are the containers which contain a relatively large number of chunks needed by the current recipe window, like Container $\#2$ in the figure, while regular containers are typically the containers having fewer chunks (i.e., the containers are fragmented), or the containers having many chunks but those chunks are accessed sparsely (i.e., sparse containers). Although leveraging the recipe knowledge to speed up restore performance has been suggested previously [6], [8], to our best, leveraging recipe to characterize containers has not been proposed in existing literatures. Second, we segregate the same restore cache into two separate caches, which aim to accommodate the two types of container, respectively. The cache for *regular* containers is managed in chunk granularity, meaning only the chunks that are required by the current recipe window are kept in the cache and other chunks are dropped off (chunk D, X, Y in Figure 4b) when the parent container is brought into the cache. The other cache for *bursty* containers is managed in container granularity as with the LRU restore cache. The sizes of the two caches can be configured dynamically according to the relative percentages of *bursty* and *regular* containers. In our experiments, we set 1/4 of the cache space for *bursty* containers. Such cache separation enables *bursty* containers to be quickly evicted from the cache, avoiding their unnecessarily occupying the cache. Particularly, it may take a reasonably long period of time for *bursty* containers to be aged in an LRU cache before they are evicted, even though none of their chunks is needed any more.

To put the mechanisms in context, Figure 4b shows how the mechanisms reduce unnecessary container reads with other assumptions being the same as in Figure 4a. After analyzing the recipe, the system knows that Container $\#2$ is a *bursty* container. So it puts Container $\#2$ in the bursty cache on the right side when bringing in Container $\#2$. Moreover, it simply discards chunks that will not be used by the current recipe window, including chunk D, X, Y, when bringing in their parent containers. As can be seen, with the two mechanisms in place, it eliminates the second access to Container $\#1$.

Bursty containers are identified based on two metrics. The first metric is the total number of chunks (denoted by N) contained in a container and required by the current recipe window. The other metric is called *weighted chunk access distance (WCAD)*, which is defined as $WCAD = \frac{\sum_{i=1}^{N-1} d_i}{N}$, where d_i denotes the i^{th} chunk access distance in unit of chunks. Assume the container size is M (chunks), then if a container satisfies the following conditions $N >= \frac{M}{2}$ and its $WCAD < \frac{M}{2}$, we claim it as a *bursty* container. Take Container $\#2$ in Figure 4 for example. Container $\#2$ contains a total number of 3 chunks (i.e., B, G, E) that are needed by the given recipe and its $WCAD$ is $\frac{1+0}{3} = \frac{1}{3}$, satisfying both $3 >= \frac{3}{2}$ and $\frac{1}{3} < \frac{3}{2}$, therefore, it is classified as a *bursty* container.

4. Evaluation

We implement *FS-CARE* in the *destor* [4] deduplication framework, an open-source platform for evaluating data deduplication techniques. It is convenient to experiment a large variety of de-duplication parameters. We aim to evaluate the effects of our proposal in reducing container reads and speeding up restore process (using the *Speed Factor* metric defined in [4]). The used workloads are listed in Table 1, which are representative of backup workloads. *Macos* and *Fslhomes* are from [16], and *Kernel* is the Linux kernel source dataset. We configure two cache sizes as 128M, 256MB, 1/4 of which is used to host *bursty containers*. The default recipe window size is set to contain 64K chunks.

TABLE 1: Workloads Statistics

Dataset name	Kernel	Macos	Fslhomes
Total size	105 GB	15.2 TB	14.8 TB
# of versions	268	83	189
Avg. chunk size	5.29 KB	128 KB	128 KB
Fragmentation	Severe	Moderate	Moderate

4.1. Container Reads

Figure 5 shows the number of container reads for restores to every backup version. We compare *LRU*, *FS-CARE*, and optimal restore cache with infinite size. The optimal cache size is used for reference. As can be seen from the left subfigure, the *FS-CARE* curve persistently lies between the *LRU* and *Opt* for all restores, demonstrating *FS-CARE* incurs less container reads than *LRU*. Please note that the sudden changes on the curves of *Fslhomes* are due to user switches, i.e., the datasets belong to different users and have different sizes. The right subfigure shows the averaged container reads across all restores for two cache sizes. The average reductions are 58.9%, 6.8%, and 15.7% with 128MB cache for *Kernel*, *Macos*, and *Fslhomes*, respectively. Those number become 45.3%, 4.6%, and 14.4% with 256MB cache.

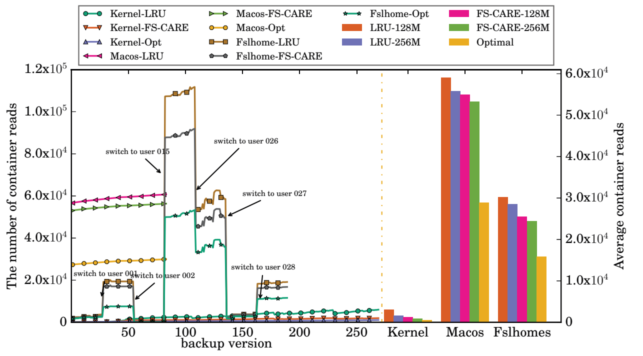


Figure 5: This figure shows *FS-CARE* is able to reduce container reads across all restores for the workloads.

4.2. Speed Factor

As a result of reductions in container reads, restore performance will be sped up. Figure 6 compares the speed factor of *LRU* and *FS-CARE*. Similar to the previous figure, the left subfigure gives the speed factors for all restores, while the right subfigure gives the geomean speed factors for each workload. *FS-CARE* improves the restore speed by up to 3.05X for *Kernel* with 128MB cache. The average improvements are 2.09X, 7.3%, 11.7% for *Kernel*, *Macos*, and *Fslhomes* with 128MB cache and the numbers become 63%, 4.8%, 9.9% with 256MB cache. It is interesting to observe that *Kernel* enjoys the most speedups due to its highest fragmentation, demonstrating that our proposal is effective in alleviating the chunk fragmentation problem in de-duplicated backup systems.

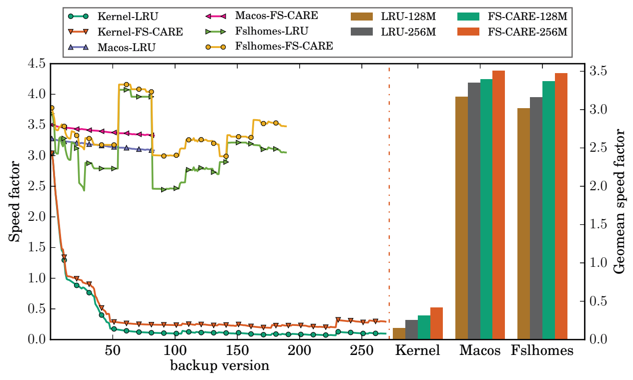


Figure 6: This figure compares speed factors. *FS-CARE* improves the restore speed relative to *LRU* cache.

4.3. The Percentage of Bursty Containers

Figure 7 shows the ratios of bursty containers for each restore. As can be seen, *FS-CARE* has identified a reasonable amount of bursty containers for each restore operation, which helps reducing container reads as we discuss in Section 3. The average values are 40.3%, 77.2%, 79.2% for *Kernel*, *Macos*, and *Fslhomes*, respectively.

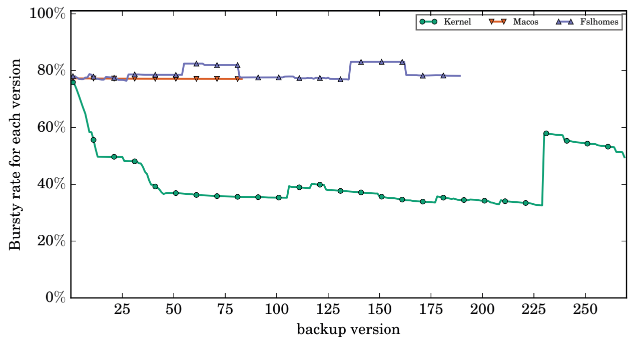


Figure 7: This figure illustrates *FS-CARE* has identified a good percentage of bursty containers for every restore.

4.4. Sensitivity On Recipe Window Size

In this section, we carry out experiments to investigate how recipe window size affects container reads and speed factor. We vary the recipe size in the range from 16K to 128K chunks. Figure 8 shows the results for *Fslhome*. It is interesting to observe that as the recipe window sizes increases, the number of container reads increases and the speed factor decreases, which seems to be counterintuitive. The intuition is that with more restore cache space, the performance improvement should be higher. We assume the reason is because every workload has an optimal recipe window size which is related to workloads characteristics, e.g., locality. We plan to further investigate this observation in the future.

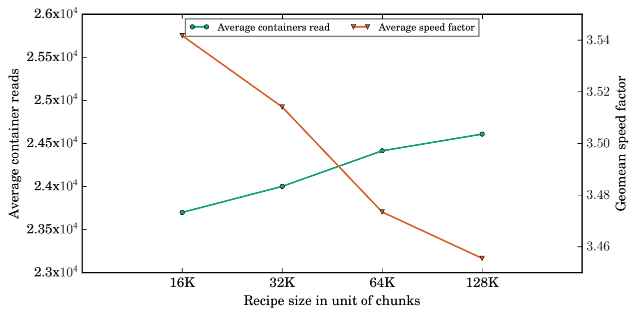


Figure 8: The effects of the recipe window size on container reads and speed factor.

5. Related Work

The chunk fragmentation problem could cause significant restore performance degradation in deduplication systems. A variety of rewriting algorithms which selectively rewrite duplicate chunks at backup time have been suggested to alleviate chunk fragmentation. Srinivasan el at. [14] address the fragmentation problem in primary de-duplication systems by performing selective de-duplication, i.e., only de-duplicating a sequence of duplicate blocks when their corresponding stored blocks are already sequentially located on disk. Kaczmarczyk et al. [5] propose another selective de-duplication approach called “context-based rewriting (CBR)”. Their idea is to maintain a logical and a physical context stream for each duplicate block. A duplicate block is de-duplicated only when the common blocks between the two contexts exceed a configured threshold limit. Lillibridge et al. [8] suggest “container capping (CAP)” to reduce chunk fragmentation. CAP divides an incoming data stream into logical fix-sized “segments” (20MB) and for each segment it sorts the containers according to the amount of fingerprints included in each container. Then CAP only eliminates the chunks whose fingerprints are included in the top-k containers, capping the number of containers to be visited during a restore operation. Fu et al. [3] propose “history-aware rewriting (HAR)” algorithm to leverage the observation that sparse containers persist across consecutive backups. HAR therefore reserves such historical information about which containers are sparse when backup finishes for the next backup’s reference. The next backup avoids de-duplicating any duplicate blocks that are also found in any of those sparse containers and rewrites them. Rewriting algorithms sacrifice deduplication efficiency for restore speed. Two previous works have also leveraged backup recipe knowledge to speed up restore process. *Forward assembly area* [8] allocates a memory buffer to temporarily buffer the out-of-order chunks and once the buffer is fully filled by in-order chunks, the whole buffer is written out. However, the recipe knowledge is leveraged in a limited way due to the limited memory buffer size. Similarly, *Limited Forward Knowledge (LFK)* [6], as the name suggests, also leverages limited recipe knowledge. Moreover, it needs additional memory to maintain the oracle information and needs to keep chunks sorted, which incurs non-trivial computation complexity. Differing from that approach, *FS-CARE* is able to leverage more recipe knowledge and does not need additional memory resources.

6. Conclusion

We present *FS-CARE*, a new segregated restore cache scheme, which aims to alleviate the restore performance slowdowns caused by chunk fragmentation. It leverages the recipe knowledge to categorize the containers into *bursty* and *regular* containers, which are put into separate caches. In doing so, *bursty* containers can get evicted from the cache at an earlier chance when they are no longer needed, improving cache efficiency. Our evaluation results have shown that our proposed cache can improve restore speed factor and reduce container reads, relative to the commonly used LRU-managed restore cache. In our future work, we plan to compare *FS-CARE* with other restore cache schemes which also leverage recipe knowledge (discussed in Section 5), under the same memory resource budget.

7. ACKNOWLEDGMENTS

The research is supported in part by the National Natural Science Foundation of China (NSFC) under grant Nos.61232004, 61502189, 61331010, and 61572012, and the U.S. National Science Foundation (NSF) under Grant Nos. CCF-1547804, CNS-1218960, and CNS-1320349. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding agencies.

References

[1] AMVROSIADIS, G., AND BHADKAMKAR, M. Identifying Trends in Enterprise Data Protection Systems. In *Proceedings of the 2015 USENIX Annual Technical(USENIX ATC'15)* (2015).

[2] EL-SHIMI, A., KALACH, R., KUMAR, A., OLTEAN, A., LI, J., AND SENGUPTA, S. Primary Data Deduplication – Large Scale Study and System Design. In *Proceedings of the 2012 USENIX Annual Technical Conference(USENIX'12)* (2012).

[3] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., HUANG, F., AND LIU, Q. Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information. In *Proceedings of the 2014 USENIX Annual Technical Conference(USENIX ATC'14)* (2014).

[4] FU, M., FENG, D., HUA, Y., HE, X., CHEN, Z., XIA, W., ZHANG, Y., AND TAN, Y. Design Tradeoffs for Data Deduplication Performance in Backup Workloads. In *Proceedings of the* 13^{th} *FAST* (2015).

[5] KACZMARCZYK, M., BARCZYNSKI, M., KILIAN, W., AND DUBNICKI, C. Reducing Impact of Data Fragmentation Caused By In-Line Deduplication. In *Proceedings of the* 5^{th} *Annual International Systems and Storage Conference(SYSTOR'12)* (2012).

[6] KACZMARCZYK, M., AND DUBNICKI, C. Reducing Fragmentation Impact with Forward Knowledge in Backup Systems with Deduplication. In *Proceedings of the* 8^{th} *International Systems and Storage Conference(SYSTOR'15)* (2015).

[7] KRUUS, E., UNGUREANU, C., AND DUBNICKI, C. Bimodal Content Defined Chunking for Backup Streams. In *Proceedings of the* 8^{th} *USENIX Conference on File and Storage Technologies(FAST'10)* (2010).

[8] LILLIBRIDGE, M., ESHGHI, K., AND BHAGWAT, D. Improving Restore Speed for Backup Systems that Use Inline Chunk-Based Deduplication. In *Proceedings of the* 11^{th} *USENIX Conference on File and Storage Technologies(FAST'13)* (2013).

[9] LILLIBRIDGE, M., ESHGHI, K., BHAGWAT, D., DEOLALIKAR, V., TREZISE, G., AND CAMBLE, P. Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality. In *Proceedings of the* 7^{th} *USENIX Conference on File and Storage Technologies(FAST'09)* (2009).

[10] LIN, X., DOUGLIS, F., LI, J., LI, X., RICCI, R., SMALDONE, S., AND WALLACE, G. Metadata Considered Harmful ... to Deduplication. In *Proceedings of the HotStorage* (2015).

[11] MEISTER, D., BRINKMANN, A., AND S$\ddot{u}\beta$, T. File Recipe Compression in Data Deduplication Systems. In *Proceedings of the* 11^{th} *USENIX Conference on File and Storage Technologies(FAST'13)* (2013).

[12] PAULO, J., AND JOS$\acute{E}$PEREIRA. A Survey and Classification of Storage Deduplication Systems. *ACM Computing Surveys 47*, 1 (May 2014), 11:1–11:30.

[13] QUINLAN, S., AND DORWARD, S. Venti: a New Approach to Archival Storage. In *Proceedings of the* 1^{st} *USENIX Conference on File and Storage Technologies(FAST'02)* (2002).

[14] SRINIVASAN, K., BISSON, T., GOODSON, G., AND VORUGANTI, K. iDedup: Latency-aware, Inline Data Deduplication for Primary Storage. In *Proceedings of the* 10^{th} *USENIX Conference on File and Storage Technologies(FAST'12)* (2012).

[15] TARASOV, V., JAIN, D., KUENNING, G., MANDAL, S., PALANISAMI, K., SHILANE, P., TREHAN, S., AND ZADOK, E. Dmdedup: Device Mapper Target for Data Deduplication. In *Proceedings of the 2014 Ottawa Linux Symposium(OLS'14)* (2014).

[16] TARASOV, V., MUDRANKIT, A., BUIK, W., SHILANE, P., KUENNING, G., AND ZADOK, E. Generating Realistic Datasets for Deduplication Analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference(USENIX ATC'12)* (2012).

[17] TURNER, V., GANTZ, J. F., REINSEL, D., AND MINTON, S. The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things. *White Paper* (2014).

[18] WALLACE, G., DOUGLIS, F., QIAN, H., SHILANE, P., SMALDONE, S., CHAMNESS, M., AND HSU, W. Characteristics of Backup Workloads in Production Systems. In *Proceedings of the* 10^{th} *USENIX Conference on File and Storage Technologies(FAST'12)* (2012).

[19] WILDANI, A., MILLER, E. L., AND RODEH, O. HANDS: A Heuristically Arranged Non-Backup In-line Deduplication System. In *Proceedings of the IEEE* 29^{th} *International Conference on Data Engineering (ICDE)* (2013).

[20] ZHU, B., LI, K., AND PATTERSON, H. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the* 6^{th} *USENIX Conference on File and Storage Technologies(FAST'08)* (2008).