2016 IEEE 22nd International Conference on Parallel and Distributed Systems

DEC: An Efficient Deduplication-Enhanced Compression Approach

Zijin Han[†], Wen Xia[‡], Yuchong Hu[†*], Dan Feng[†], Yucheng Zhang[†], Yukun Zhou[†], Min Fu[‡], Liang Gu[‡]
[†]*Huazhong University of Science and Technology*
[‡]*Sangfor Technologies Co., Ltd.*
[*]*Corresponding author: yuchonghu@hust.edu.cn*

***Abstract*—Data compression is widely used in storage systems to reduce redundant data and thus save storage space. One challenge facing the traditional compression approaches is the limitation of compression windows size, which fails to reduce redundancy globally. In this paper, we present DEC, a Deduplication-Enhanced Compression approach that effectively combines deduplication and traditional compressors to increase compression ratio and efficiency. Specifically, we make full use of deduplication to (1) accelerate data reduction by fast but global deduplication and (2) exploit data locality to compress similar chunks by clustering the data chunks which are adjacent to the same duplicate chunks. Our experimental results of a DEC prototype based on real-world datasets show that DEC increases the compression ratio by 20% to 71% and speeds up the compression throughput by 17%∼183% compared to traditional compressors, without sacrificing the decompression throughput by leveraging deduplication in traditional compression approaches.**

***Keywords*-Traditional Compression, Data Deduplication, Data Locality, Storage Systems**

I. Introduction

With the growing amount of data worldwide in recent years, data reduction technologies have attracted extensive attention. Compression, a mainstream technology of data reduction, encodes the frequently appeared information into fewer bytes than its original form [1], [2]. Traditional compression approaches are mostly lossless compression, reducing data based on byte-level Huffman coding [3] and string-level dictionary coding [1], [4]. Lossless compression approaches, such as gzip [2] and bzip2 [5], detect redundant data in a sliding window under consideration of compressing time and the overhead of memory. This leads to suboptimal compression performance because redundancy among the different compression windows cannot be eliminated.

Data deduplication [6]–[8] is another kind of data reduction technology. Different from traditional compression approaches, data deduplication reduces redundant data in chunk-level or file-level. Generally speaking, a chunk-level data deduplication scheme divides data stream into multiple chunks, and then uniquely identifies these chunks by a secure SHA-1 or MD5 hash signature which is also called fingerprint [9]. Duplicate chunks can be detected by matching fingerprint, which avoids the complicated bytes-matching in the traditional compression approaches. Hence, the chunk-level deduplication approaches have higher throughput and better scalability than traditional compression approaches, and thus have been widely used in backup and archiving storage systems [10], [11].

However, data deduplication can only identify and remove completely duplicate data files and chunks while fails to detect redundancy among non-duplicate but very similar files and chunks. Delta compression, on the other hand, solves the problem [12]–[14]. Generally, delta compression is able to eliminate redundancy between similar chunks by quickly matching the duplicate strings, but at the cost of additional computation and memory overheads.

A lot of studies like SIDC [13], DARE [15], and Ddelta [16] suggest that combining the traditional compression and delta compression technologies on top of deduplication can maximize the data reduction. Especially in DARE [15], it presents a method that exploits the information of duplicate-adjacency to detect resemblance (similar data) for further delta compression. Inspired by this, we believe that *it is possible to improve the compression performance by also exploiting duplicate-adjacency information*. In other words, we aim to cluster data around the same duplicate chunks to solve the problem of local restriction on traditional compression window, which can not eliminate the redundancy across different windows.

In this paper, we propose DEC, an efficient Deduplication-Enhanced Compression approach that "exploits" deduplication to improve the performance of traditional compression. To the best of our knowledge, it is the first paper that uses deduplication to optimize traditional compression performance. Specifically, making full use of data deduplication for traditional compression mainly has two advantages:

1) Data deduplication can quickly eliminate redundancy globally at chunk-level, where the chunking and fingerprinting speed is much faster than traditional compressors. In addition, the secure fingerprinting technique also simplifies the process of identifying duplicate chunks globally.
2) Duplication-enhanced clustering method, which regards the chunk as similar if their respective neighbors are duplicates, can eliminate redundancy among

1521-9097/16 $31.00 © 2016 IEEE
DOI 10.1109/ICPADS.2016.73

Table I
TRADITIONAL LOSSLESS COMPRESSION VS. DATA DEDUPLICATION.

	Lossless Compression	Data Deduplication
Object	All data	Duplicate data
Granularity	Bytes/Strings	Chunks/Files
Compression Rage	Local	Global
Representative Prototypes	GZIP [2], 7Z [20]	LBFS [6], Venti [9]

similar chunks through clustering them in the same compression window and thus further improves the compression performance (i.e., compression ratio and speed).

Our experimental evaluation of the DEC prototype, driven by several real-world workloads, shows that DEC increases the compression ratio of traditional compressors by 20%∼71% and speeds up compression by 17%∼183% while achieving a comparable decompression speed.

The rest of paper is organized as follows. Section II presents the background and motivation of this paper. We detail design and implementation of DEC in Section III. We discuss the experimental evaluation of DEC in Section IV. Section V draws conclusions and outlines future work.

II. BACKGROUND AND MOTIVATION

A. Background

Traditional lossless compression approaches eliminate redundancy at byte/string level by performing Huffman coding [3] and dictionary coding [1] (e.g., gzip [2]). Most compression approaches detect and eliminate redundant strings in a data window called sliding window. The larger window size means the greater chance to find redundant strings, leading to a higher compression ratio. However, with the increasing of the window size, the overhead of finding redundant strings will be increased exponentially, so most compression approaches set a limit to the window size for trading off compression ratio and speed. For example, the window size of DEFLATE used by gzip is 64KB, bzip2 has a maximum window of 900KB [17]–[19]. Redundant data which are not in the same sliding window can not be eliminated. Therefore, traditional lossless compression approaches can not eliminate redundancy globally in storage systems nowadays due to high overhead of computation and weak scalability.

Since traditional data compression can not satisfy the need of global data reduction in storage systems, data deduplication, a chunk-level data reduction technology, rose in response to the proper time. Deduplication splits data into chunks by Content-Defined Chunking (CDC) [21] (e.g., Rabin-based Chunking) and then identifies these chunks by their secure fingerprints. Deduplication regards the chunks as duplicate chunks if their fingerprints match.

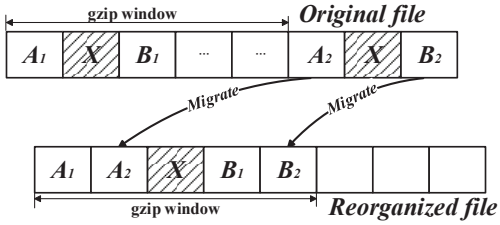


Figure 1. An example of deduplication-enhanced compression (DEC). X represents duplicate chunks and A1, B1 are similar to A2, B2 respectively. DEC gathers similar chunks into a compression window (e.g., a gzip window) and reorders the file for better compression performance.

Table I compares traditional lossless compression with data deduplication. Generally, data deduplication is faster than traditional compression for two reasons: (1) it quickly matches the duplicate chunks by chunking and fingerprinting and (2) it reduces redundancy at a much higher granularity. Traditional compressors limit the range of finding redundancy, but deduplication mitigates this drawbacks by searching duplicate chunks globally. Data deduplication has been widely used in commercial backup and archiving systems due to its higher scalability and efficiency. For example, Data Domain [7], HP [22], IBM [23] focused on using deduplication to promote backup system performance.

Currently, one challenge facing deduplication is that there is still a lot of redundancy among the non-duplicate but very similar data after deduplication. It is considered essential to eliminate redundancy among similar chunks/files for further saving storage space. Traditional compression approaches have the ability to remove redundancy among similar data files and chunks, so it is regarded as a complement to deduplication approaches. Therefore, our paper aims at combining data deduplication and traditional compression, where deduplication will enhance the compression effectiveness but also promote the compression speed.

It is worth noting that, varying from delta compression [14] and Migratory Compression [17], DEC does not compute features for resemblance detection. It only explores the existing deduplication information to enhance the traditional compression performance, by regarding data chunks adjacent to duplicate chunks as similar chunks according to the content locality.

B. Motivation

In order to eliminate the redundancy among the non-duplicate but very similar chunks/files, resemblance detection is required in the traditional delta compression approaches. DARE, a deduplication-aware resemblance detection scheme, combines data deduplication and delta compression. It employs a scheme by regarding chunks to be similar if their adjacent chunks are duplicate, and further uses an improved super-feature approach to enhance the efficiency of resemblance detection [15], [24].

Inspired by DARE, we propose a duplication-enhanced

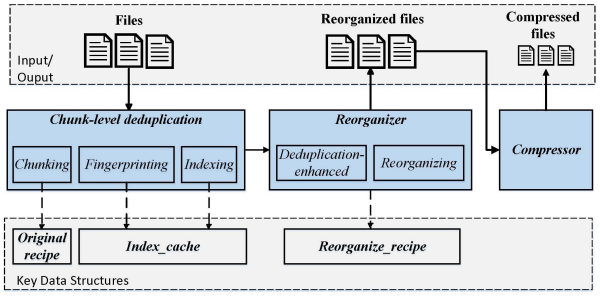


Figure 2. The compression workflow of DEC.

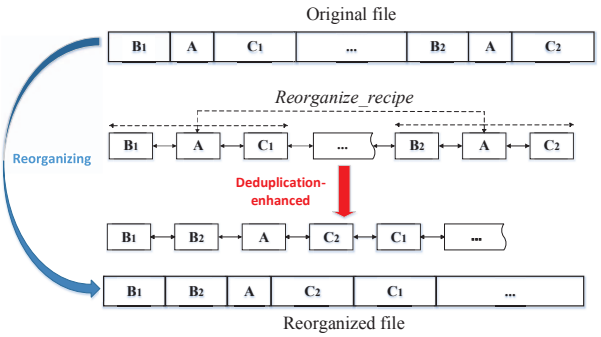


Figure 3. The process of *deduplication-enhanced* and *reorganizing*.

compression approach (i.e., DEC). Figure 1 shows that if chunk X is duplicate, DEC will consider chunks around X as potentially similar chunks (i.e., A1&A2 and B1&B2), and then migrate the similar chunks into one compression window (e.g., gzip compression window) in order to remove redundancy as much as possible. DEC has two advantages compared to the traditional compressors as follows.

- DEC has a high speed of compressing because it detects resemblance on basis of deduplication-enhanced instead of extra resemblance detection. In addition, similar data gathered in one sliding window will make it easier to compress, resulting in a shorter time for compression.
- DEC solves the problem of local restriction on compressing window. On one hand, deduplication removes duplicate chunks all over the files. On the other hand, duplication-enhanced migrates the non-duplicate but similar chunks into one compressing window to maximize the redundancy elimination. Both of them break through the limit of sliding window size in traditional compressors, making it possible to globally reduce redundancy in storage systems.

In all, the motivation of this work is to leverage the deduplication to enhance the performance of traditional compressors, including the compression ratio and speed.

III. Design and Implementation

A. Compression

The goal of DEC is to improve local-based redundancy removal of traditional compressors. As shown in Figure 2, compression workflow of DEC consists of three parts: chunk-level deduplication, reorganizer, and compressor.

1) Chunk-level deduplication: Chunk-level deduplication in DEC mainly takes three key steps: chunking, fingerprinting, indexing.

Chunking: Chunking algorithms take charge of splitting data into fixed size or variable size chunks. Fix-Sized Chunking (FSC) [9], [25] is simple and fast. However, FSC has a disadvantage of low deduplication efficiency caused by the *boundaries-shift* problem. For example, if we insert a byte at the beginning of the file, all the chunk boundaries marked by FSC will be shifted, and duplicate chunks will be changed [26]. Content-Defined Chunking [6] solves the *boundary-shift* problem by finding cut-point depending on local content. Moreover, chunking algorithms play an important part in redundancy detection and deduplication-enhanced clustering method later. In this paper, we adopt a recent proposed fast and efficient CDC approach, called Asymmetric Extreme (AE) chunking [26] instead of the traditional Rabin-based chunking [6]. We employ *original_recipe* to record chunks' sequence and location in the file after chunking, which is used for restoring the file in decompression.

Fingerprinting: Each chunk is fingerprinted by SHA-1 later. Fingerprinting is employed to uniquely identify the chunk as its literal meaning. If the chunks are duplicate, their fingerprints will be identical.

Indexing: We use an in-memory hash table *index_cache* for deduplication indexing. Fingerprints will be searched in *index_cache*. If they have been found, DEC will update the *original_recipe*; If not, the fingerprints will be recorded in *index_cache* at the first time and return the information of their locations in data stream. Chunk-level deduplication only delivers non-duplicate chunks to reorganizer while duplicate chunks are removed.

The overhead of chunk-level deduplication module mainly lies in AE chunking and fingerprinting. The time overheads for chunking and fingerprinting are very low compared to the traditional compressing, which will be studied and evaluated in Section IV. We store the information of original chunk sequence and fingerprints respectively in *original_recipe* and *index_cache* which are in memory. For example, *original_recipe* will take up 16MB and *index_cache* costs 5MB of memory space for a 3GB file. Therefore, the storage overhead is acceptable.

2) Reorganizer: The module of reorganizer consists of *deduplication-enhanced* and *reorganizing*. Redundant data has the feature of spatial locality, which means that if chunks A, B, and C appeared in sequence once, chunks B and C will be probably around chunk A for the next time. Inspired by this, we may consider any two chunks to be similar if their respective adjacent data chunks are

duplicate [15]. DEC senses duplicate-adjacency chunks and then clustering them to reorganize a new file. *Deduplication-enhanced* method exploits the existing duplicate-adjacency information to find similar data, thus reducing the overhead of similarity detection. In order to get the location of duplicate chunks easily, we employ a doubly-linked list– *reorganize_recipe* to record the metadata information (i.e., offset, length, fingerprint, etc.) of chunks. Figure 3 shows the process of *deduplication-enhanced* and *reorganizing*. Similar chunks will be found if DEC detects duplicate chunks and then traverses the *reorganize_recipe* by a doubly linked list. *Deduplication-enhanced* method gathers the potential similar chunks together for better compression performance. When DEC traverses *reorganize_recipe* from head to tail, it will get the chunk sequence of reorganized file.

Reorganizer gathers *original_recipe* and data to generate a new reorganized file. Figure 4 illustrates the format of reorganized file. Reorganizer combines job information of the program with *original_recipe* and write them in reorganized file as *metadata*. According to *reorganize_recipe*, DEC reads non-duplicate data chunks from file and writes them in reorganized file as *data* segment. DEC updates the offsets of chunks recorded in *original_recipe* with offsets in reordered file so that DEC can locate the data with *original_recipe* when restoring.

In general, the overheads for the reorganizer are twofold:

- *Memory overhead*: In the process of *deduplication-enhanced* approach, the cost of memory space is quite low because *reorganize_recipe* only records the logic sequence of chunks instead of data. Each chunk information will be associated with two pointers for building the doubly-linked list, only costing about 8 or 16 bytes, which is acceptable.
- *Computation overhead*: The *deduplication-enhanced* approach exploits the existing information of deduplication in memory for resemblance detection, which brings no extra computation overhead. The actual extra computation of overhead occurs when file is stored in disk. Reading data chunks will access disk randomly, thus costing a large number of I/Os. We employ the 'multi-pass' algorithm mentioned in MC [17]. DEC scans the file repeatedly from start to end and buffers the data chunks which need to be written in the reorganized file. Data chunks in memory buffer are output to the reorganized file at the end of each pass. For the reason that file is stored in memory in our experiments, DEC only needs to scan the original file for once.

Based on above discussions, the overhead for detecting resembling chunks in DEC is very low, which will be also demonstrated in Section IV.

3) Compressor: Finally, DEC compresses the reorganized files by using the traditional compressors, such as gzip or bzip2. Similar data has been gathered together after processing of reorganizer, leaving them in the same sliding window so that the off-the-shelf compressor is able to eliminate redundancy among similar chunks (because the sliding window is much larger than the chunk sizes). For the traditional compressor, we consider gzip [2], bzip2 [5], etc. and run the compressors with the default parameters in DEC. We compare the results of DEC with these compressors separately and it turns out that DEC is much better than traditional compression approaches (See section IV).

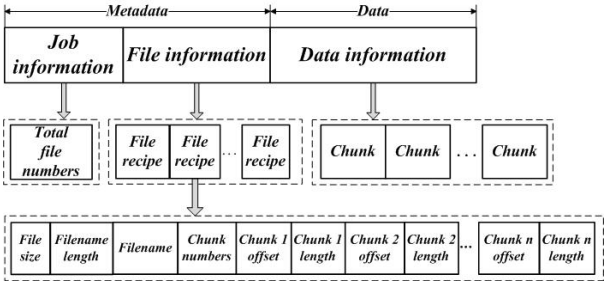


Figure 4. The format of reorganized file.

B. Decompression

Decompression takes only two steps to restore the original file: decompressing and restoring. First of all, DEC will decompress the compressed file by corresponding decompressors with the default parameters. Then restore module extracts *original_recipe* from *metadata* in decompressed file and restores data chunks sequence with the help of the *original_recipe* to generate the original file.

When restoring the file, no more memory overhead is needed except for a list to store *original_recipe* because all the data chunks store in memory already. The RAM memory overhead is negligible. However, when files are stored in disk, DEC needs a buffer to read and store data from disk consecutively. Restoring the file will also access disk randomly, leading to more I/O overhead.

IV. Performance Evaluation

A. Experimental Setup

1) Experimental Platform: We implement the DEC prototype on Ubuntu 14.04 operating system running on a 4-core Intel i7-4770 processor at 3.4GHz with 16GB memory and 2TB hard disk. GCC version 4.8.2 is used to compile the code with the compiler option "-O3".

2) Experimental Metrics: We consider compression ratio, compression and decompression throughput as three important metrics in DEC. *Compression ratio is the ratio of the data size after and before compression*, so lower compression ratio represents saving more storage space. *Compression throughput and decompression are all recorded by dividing the original size (i.e., before compression) by the system running time*. In general, throughput shows data size processed by compressor per second.

3) Experimental Configurations: We choose AE chunking algorithm [26] as content-defined chunking and SHA-1 hash function for fingerprinting in the data deduplication stage of DEC. We employ the "in-memory" approach which means we put the original file in memory and ignore I/Os, as that in MC [17] and the traditional compressors. To evaluate the influence of varying parameters in DEC, we set a number of parameters that can be comprehensively studied:

- *Chunking algorithm*: Fixed-size chunking or AE chunking algorithm where AE is the default;
- *Chunk size*: We study the chunk sizes for AE chunking of 2, 4, 8, 16, 32, and 64KB where the default chunk size is 8KB;
- *Compressors*: four classical compressors, gzip, bzip2, 7z, and rar, are selected for our study.

4) Datasets: Two kinds of source code files and virtual machine images are used in evaluating the performance of DEC, which represents the typical compression workloads in storage systems. The source code files are tarred by versions for simplifying the compression process. Deduplication ratio means the percentage of duplicate data and it is tested in DEC using AE chunking algorithm with an average chunk size of 8KB. The characteristics of four datasets are as follows.

- We use glib and gdb as the datasets of source code files. Glib consists of 35 different versions and its size is about 3.0GB with 53% duplicate data. As for gdb dataset of about 2.0GB, it is composed of 15 versions and has a deduplication ratio of 55%.
- Debian and ubuntu are virtual machine disk images, both consisting of several VMDK files. Debian is about 2.1GB and its deduplication ratio is 28% while those of Ubuntu are about 1.7GB and 26%, respectively.

B. A Sensitivity Study of DEC

In order to help understand the design of DEC, we first study the runtime breakdown of DEC and then see the influence of different parameters on performance of DEC.

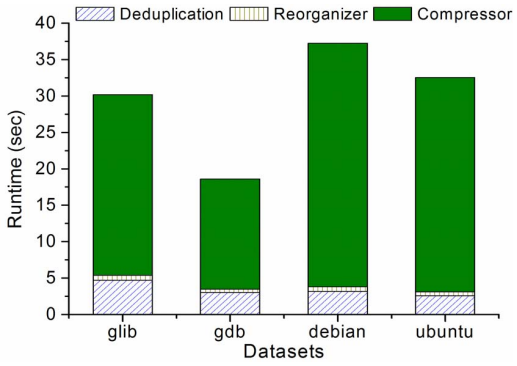


Figure 5. Runtime breakdown of DEC on four datasets. DEC employs the default configuration: AE algorithm, an average chunk size of 8KB, and gzip compressor.

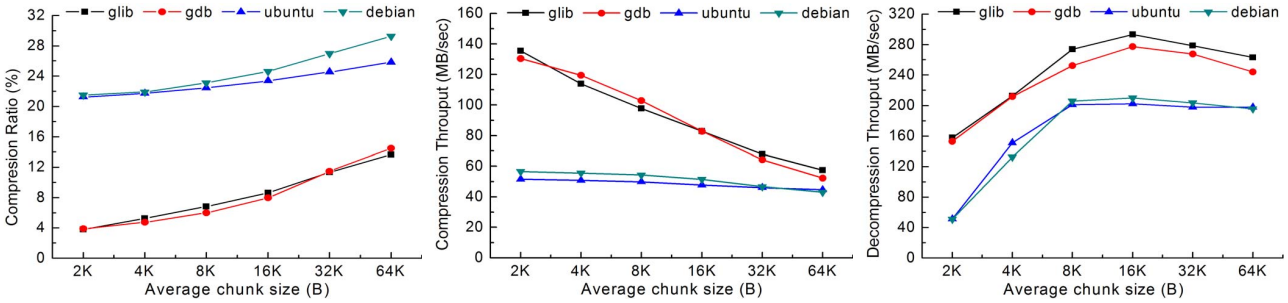
Table II
RUNTIME OF MODULES IN DEC ON FOUR DATASETS.

Time	Dataset			
Module	Glib	Gdb	Debian	Ubuntu
Chunking	1.13s	0.74s	0.77s	0.62s
Fingerprinting	3.60s	2.30s	2.43s	1.96s
Deduplication-enhanced	0.09s	0.08s	0.10s	0.08s
Reorganizing	0.58s	0.38s	0.51s	0.47s
Compressor	24.80s	15.14s	33.45s	29.40s
Total	30.20s	18.64s	37.26s	32.53s

1) Runtime Breakdown: Figure 5 shows the runtime breakdown of DEC on four datasets. Here DEC is configured with the default parameters: AE chunking algorithm, average chunk size of 8KB, and gzip compressor. Table II shows the runtime of modules on four datasets. First of all, Deduplication phase takes from 8% to 16% of overall time. In fact, AE chunking in chunk-level deduplication is fast enough and its speed can reach about 2.7GB per second. Fingerprinting costs most of time in the deduplication phase but it is not the bottleneck of DEC. Second, Reorganizer takes less than one second on all four datasets as shown in Table II because it uses locality to sense similar chunks and thus simplifies the resemblance detection. Because all files are tested in memory, reorganizing a file does not spend much time. Last but not least, Figure 5 also suggests that compressor module takes above 80% of overall time on the four datasets, being the main bottleneck in DEC. Different compressors cost different time, which is due to their detailed encoding algorithms. In general, the deduplication and "deduplication-enhanced" approaches used in DEC only account for a small fraction of the total time overheads while greatly improving the final compression performance as shown in Section IV-C.

2) Chunk size variance: Chunk size is an important parameter in DEC. As shown in Figures 6(a) and 6(b), compression ratio increases and compression throughput decrease as the chunk size increases. It is because DEC detects less duplicate chunks when using larger average chunk size and then spending more time on gzip compressor. Meanwhile, detecting less duplicate chunks means less similar chunks will be detected by "locality-enhanced", and thus less redundancy eliminated by DEC. In addition, the compression throughput differs much on the four datasets, which is also due to the different deduplication ratio of the four datasets.

As shown in Figure 6(c), decompression throughput increases dramatically with chunk size increasing before 8KB but fluctuates little after 8KB. This is because, when DEC detects more duplicates using smaller average chunk size, more duplicate-adjacent chunks are regarded as similar chunks, and thus DEC requires more time on decompressing and reorganizing those similar chunks by the traditional compressors. Since the decompression throughput stays almost unchanged after 8KB, we set 8KB as the default



(a) Compression ratio (b) Compression throughput (c) Decompression throughput

Figure 6. Compression performance of DEC as a function of the average chunk size. Here DEC employs the AE for chunking and gzip as compressor.

average chunk size used in the following evaluation.

3) Chunking algorithm: Figure 7 compares AE chunking and FSC with an average chunk size of 8KB in DEC. In general, AE is better than FSC in all three metrics. On the metric of compression ratio, comparing to DEC with FSC, DEC with AE chunking reduces 19%∼26% more data in VM datasets and 55%∼62% more in gdb and glib datasets. On the metric of compression throughput, comparing to the DEC with FSC, DEC with AE chunking is 18%∼29% faster in VM datasets and 92%∼120% faster in the other two datasets. In addition, DEC with AE chunking also performs well on the decompression metric. In a word, AE chunking is much better than FSC in DEC by addressing the boundary-shift problem and thus detecting more redundancy. Therefore, we set AE chunking as the default chunking algorithm in DEC.

C. Putting It All Together

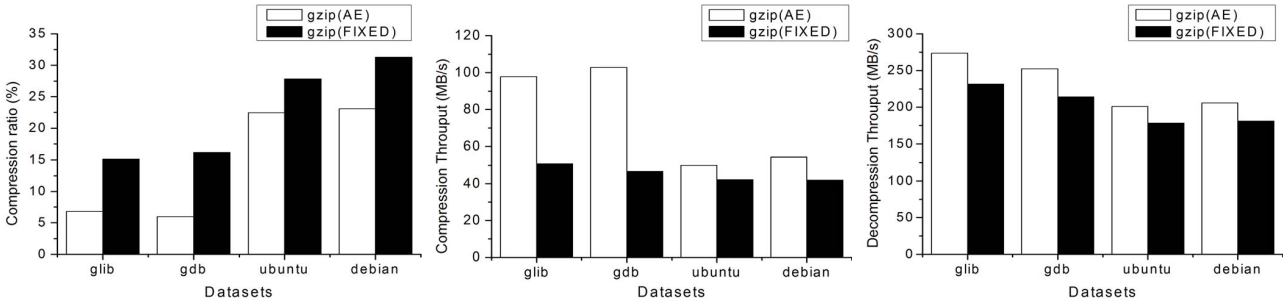
In this subsection, we test the overall performance of DEC, comparing it to traditional compressors and the way that directly combining deduplication and compression to see how much DEC can promote the performance of traditional compressors. And based on the above experimental evaluation, we set AE chunking algorithm and an average chunk size of 8KB as the default configuration of DEC.

Figure 8 shows compression ratio of traditional compressors, combination of deduplication and compression, and DEC. In general, DEC detects more redundancy regardless of the compressors. It can eliminate 67%∼71% more data than the traditional approaches in glib and gdb datasets. DEC can also enhance compression ratio from 20% to 33% in virtual machine images datasets. In addition, compared to the method which combines deduplication and traditional compressors, the promotions are still obvious. The promotion of compression ratio performs well on open source datasets (from 19% to 36%) but only slightly better (from 2% to 5%) than combining deduplication and compressions on VM datasets. The main reason for low promotion on VM datasets is: few similar chunks are detected due to low deduplication ratio on these datasets.

Figure 9 shows compression throughput of four compression approaches. DEC achieves speedups of about 1.07X to 2.28X over the traditional compressors on gdb and glib, but 17%∼41% on ubuntu and debian. Compared to the method of combining deduplication and compression, DEC only increases the compression throughput by 8%∼26% on gdb and glib datasets and almost identical (i.e., from 0.7% to 5.6%) except for bzip2 on the VM dataset. From figure 9, we draw the conclusion that the high speed of compression in DEC mainly relies on data deduplication technique.

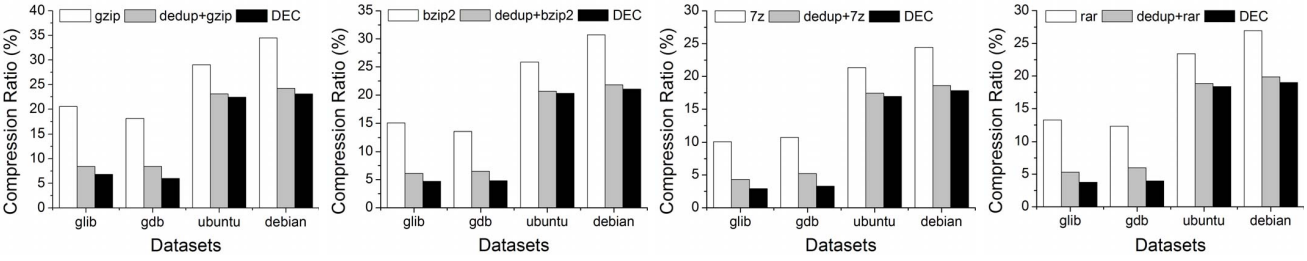
Figure 10 shows decompression throughput of four evaluated approaches. Comparing to the traditional compressor on the metric of decompression throughput, DEC can promote 12%∼114% with bzip2 and 7z compressors but obtains nearly the same and sometimes lower results with gzip and rar. This is because gzip and rar decompress fast, but sometimes restoration of DEC will cost more overhead so that DEC will be slower than the traditional compression. Therefore, we can conclude that DEC can shorten the decompression time with the traditional compressors which compress better (eg., bzip2, 7z). In comparison with the method of compressing after deduplication, DEC has an advantage of 8%∼27% on open-source datasets but no obvious promotion on virtual machine images datasets for the same reason of low deduplication ratio.

To sum it up, DEC exploits deduplication to improve the performance of traditional compressor, achieving the following optimizations. First of all, it eliminates duplicate data through deduplication technology and reduces similar data by clustering duplicate-adjacency in one compression window leading to running faster and saving more space. The higher the duplication ratio is, the more redundancy is eliminated. Secondly, the characteristics of deduplication technology significantly promote the speed and usage ratio of traditional compression approaches. Last but not least, the decompression throughput will be promoted on datasets which have a high deduplication ratio and stay almost unchanged even if the datasets do not have much more duplicates.



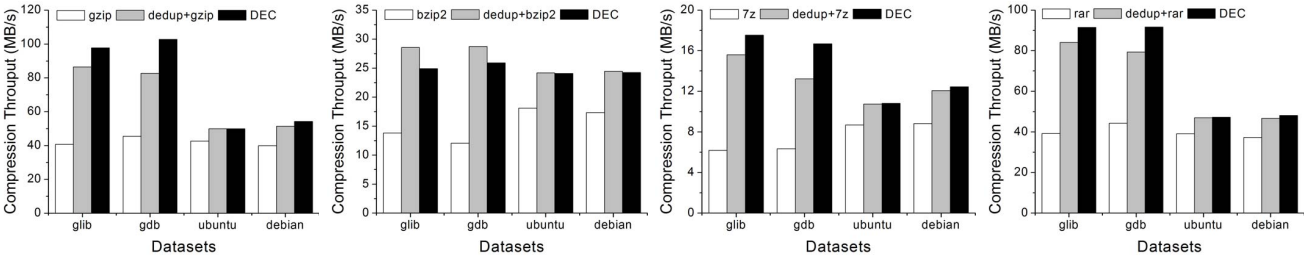
(a) Compression ratio (b) Compression throughput (c) Decompression throughput

Figure 7. Content-Defined Chunking (i.e., AE) vs. Fixed-Size Chunking. Here DEC employs average chunk size of 8KB and gzip compressor.



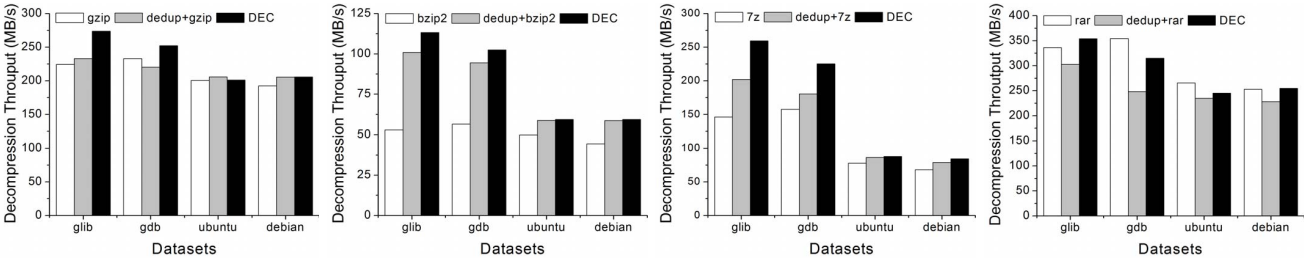
(a) Gzip (b) Bzip2 (c) 7z (d) Rar

Figure 8. Comparison of the compression ratio among the four compression approaches.



(a) Gzip (b) Bzip2 (c) 7z (d) Rar

Figure 9. Comparison of compression throughput among the four compression approaches.



(a) Gzip (b) Bzip2 (c) 7z (d) Rar

Figure 10. Compression of decompression throughput among the four compression approaches.

V. Conclusion and Future Work

In this paper, we present DEC, a deduplication-enhanced compression scheme that effectively explores deduplication to improve compressing performance of the traditional compressors. DEC fully exploits the locality of duplicate data for resemblance detection and clusters the duplicate-adjacent (i.e., potentially similar) data in a compression window to eliminate more redundancy. Our experimental results driven by four real-world datasets suggest that, comparing with the traditional compressors, such as gzip, bzip2, etc., DEC achieves the speedup at 17%∼183% and increases

the compression ratio by 20%∼71% without sacrificing the decompression performance.

In our future work, we will employ the DEC in large-scale storage systems on HDDs or SSDs, to see its potential system benefits and also the bottlenecks. In addition, we plan to further improve the performance of file reorganization and data-restoration. Our experiments also suggest that decompression throughput needs to be improved especially when there is not too much duplicate data in files. A new strategy of organization and restoration may help promote the performance of decompression.

ACKNOWLEDGMENT

This work was partly supported by the National Natural Science Foundation of China under Grant No. 61502190, 61502191, and CCF-Tencent Open Fund 2015. This work also supported by Key Laboratory of Information Storage System, Ministry of Education, China, and Sangfor Technologies Co., Ltd. The authors are grateful to anonymous reviewers.

REFERENCES

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

[2] J.Gailly and M.Adler, "The gzip compressor," http://www.gzip.org/, 1991.

[3] D. A. Huffman *et al.*, "A method for the construction of minimum-redundancy codes," *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.

[4] 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, Sept 2016.

[5] J. Seward, "Bzip," http://www.bzip.org/, 1996.

[6] A. Muthitacharoen, B. Chen, and D. Mazieres, "A low-bandwidth network file system," in *Proc. ACM SIGOPS, 2001*.

[7] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the disk bottleneck in the Data Domain Deduplication File System." in *Proc. USENIX FAST, 2008*.

[8] Y. Zhou, D. Feng, W. Xia, M. Fu, F. Huang, Y. Zhang, and C. Li, "Secdep: A user-aware efficient fine-grained secure deduplication scheme with multi-level key management," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[9] S. Quinlan and S. Dorward, "Venti: A new approach to archival storage." in *Proc. USENIX FAST, 2002*.

[10] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems." in *Proc. USENIX FAST, 2012*.

[11] C. Constantinescu, J. Glider, and D. Chambliss, "Mixing deduplication and compression on active data sets," in *Proc. IEEE DCC, 2011*.

[12] P. Kulkarni, F. Douglis, J. D. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files." in *Proc. USENIX ATC, 2004*.

[13] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "WAN-optimized replication of backup datasets using stream-informed delta compression," *ACM Transactions on Storage (TOS)*, vol. 8, no. 4, p. 13, 2012.

[14] J. J. Hunt, K.-P. Vo, and W. F. Tichy, "Delta algorithms: An empirical analysis," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 7, no. 2, pp. 192–214, 1998.

[15] W. Xia, H. Jiang, D. Feng, and L. Tian, "Combining deduplication and delta compression to achieve low-overhead data reduction on backup datasets," in *Proc. IEEE DCC, 2014*.

[16] W. Xia, H. Jiang, D. Feng, L. Tian, M. Fu, and Y. Zhou, "Ddelta: A deduplication-inspired fast delta compression approach," *Performance Evaluation*, vol. 79, pp. 258–272, 2014.

[17] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace, "Migratory compression: Coarse-grained data reordering to improve compressibility," in *Proc. USENIX FAST, 2014*.

[18] L. P. Deutsch, "DEFLATE compressed data format specification version 1.3," 1996.

[19] J. Gilchrist, "Parallel data compression with bzip2," in *Proceedings of the 16th IASTED international conference on parallel and distributed computing and systems*, vol. 16, 2004, pp. 559–564.

[20] http://www.7-zip.org/, September 1991, 7-zip.

[21] A. Z. Broder, "Some applications of Rabins fingerprinting method," in *Sequences II*. Springer, 1993, pp. 143–152.

[22] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse indexing: Large scale, inline deduplication using sampling and locality." in *Proc. USENIX FAST, 2009*.

[23] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in *Proc. ACM SYSTOR, 2009*.

[24] W. Xia, H. Jiang, D. Feng, and L. Tian, "DARE: A deduplication-aware resemblance detection and elimination scheme for data reduction with low overheads," *IEEE Transactions on Computers*, vol. 65, no. 6, pp. 1692–1705, 2016.

[25] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: a capacity-optimized ssd cache for primary storage," in *Proc. USENIX ATC, 2014*.

[26] Y. Zhang, H. Jiang, D. Feng, W. Xia, M. Fu, F. Huang, and Y. Zhou, "AE: An asymmetric extremum content defined chunking algorithm for fast and bandwidth-efficient data deduplication," in *Proc. IEEE INFOCOM, 2015*.