JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 1103-1120 (2017)
DOI: 10.6688/JISE.2017.33.5.1

Memory Deduplication: An Effective Approach to Improve the Memory System*

YUHUI DENG[1,2], XINYU HUANG[1], LIANGSHAN SONG[1],
YONGTAO ZHOU[1] AND FRANK WANG[3]
[1]*Department of Computer Science*
Jinan University
Guangzhou, 510632 P.R. China
E-mail: tyhdeng@jnu.edu.cn; huangxinyu@tisson.cn; 710260037@qq.com; y.t.zhou@foxmail.com
[2]*Key Laboratory of Computer System and Architecture*
Chinese Academy of Sciences
Beijing, 100190 P.R. China
[3]*School of Computing*
University of Kent
Canterbury, CT2 7NZ, UK
E-mail: frankwang@ieee.org

Programs now have more aggressive demands of memory to hold their data than before. This paper analyzes the characteristics of memory data by using seven real memory traces. It observes that there are a large volume of memory pages with identical contents contained in the traces. Furthermore, the unique memory content accessed are much less than the unique memory address accessed. This is incurred by the traditional address-based cache replacement algorithms that replace memory pages by checking the addresses rather than the contents of those pages, thus resulting in many identical memory contents with different addresses stored in the memory. For example, in the same file system, opening two identical files stored in different directories, or opening two similar files that share a certain amount of contents in the same directory, will result in identical data blocks stored in the cache due to the traditional address-based cache replacement algorithms. Based on the observations, this paper evaluates memory compression and memory deduplication. As expected, memory deduplication greatly outperforms memory compression. For example, the best deduplication ratio is 4.6 times higher than the best compression ratio. The deduplication time and restore time are 121 times and 427 times faster than the compression time and decompression time, respectively. The experimental results in this paper should be able to offer useful insights for designing systems that require abundant memory to improve the system performance.

Keywords: memory deduplication, address-based cache, content-based cache, memory compression, data characteristics

1. INTRODUCTION

Memory hierarchy is designed to leverage data access locality to improve the performance of computer systems. Each level in the hierarchy has higher speed, lower latency, and smaller size than lower levels. Over the past decades, the memory hierarchy has suffered from significant bandwidth and latency gaps among processor, RAM, and disk drive [1, 2]. For example, the performance of processors has continued to double

Received February 1, 2016; revised February 28, 2016; accepted March 31, 2017.
Communicated by Cho-Li Wang.
* A preliminary version of this paper appears in Proceedings of the 8th IEEE International Conference on NAS 2013.

about every 18 months since the number of transistors on a chip has increased exponentially in accordance with Moore's law. The advent of multi-core processors will further facilitate this performance improvement. Unfortunately, disk access time was improved only about 8% per year [3], although the internal data transfer rate has been growing at an exponential rate of 40% each year over the past 15 years [4]. The performance gap between processor and RAM has been alleviated by fast cache memories. However, the performance gap of RAM to disk drive has been widened to six orders of magnitude in 2000 and will continue to widen by about 50% per year [5]. Therefore, a lot of research efforts have been invested in alleviating this gap [4, 6, 7].

Many programs require more RAM to hold their data than a typical computer has[8, 9]. Although the amount of RAM in a typical computer has significantly increased due to the declining prices, program developers have even more aggressively increased their demands [10]. Programs that run entirely in RAM benefit from the improvements of processor performance, but the runtime of programs that page or swap is likely to be dominated by the disk access time when the amount of physical RAM is less than what the programs require [11, 12]. Additionally, due to the development of cloud technology, lightweight laptops are widely used as netbooks which are memory constrained but have rich CPU capability available [13].

The traditional cache replacement algorithms including Least Frequently Used (LFU), Least Recently Used (LRU), *etc.* are all address-based approaches. These methods determine which memory page should be replaced by checking whether the address of that page has been accessed. This results in many identical memory pages stored in the cache, thus decreasing the effectiveness of the memory. The reason behind this is because there are many data blocks that have identical contents but associate with different addresses. For example, in the same file system, opening two identical files stored in different directories, or opening two similar files that share a certain amount of contents in the same directory, will result in identical data blocks stored in the cache due to the traditional address-based cache replacement algorithms. Another typical example is that multiple programs call the same static link library.

Memory compression has been investigated by a lot of research efforts [11, 12, 14-17]. The basic idea is reserving some memory space that would normally be used directly by programs, compressing relatively unused memory pages, and storing the compressed pages in the reserved space. By compressing those pages, the effective memory size available to the programs becomes larger, and some memory paging and swapping can be avoided, thus eliminating some expensive disk accesses. Because accessing compressed memory is faster than accessing disk drives, memory compression can over commit memory space without significantly reducing performance. For example, when a virtual page needs to be swapped, this page can be first compressed and then maintained in the memory. When the page is needed again, it is decompressed and given back. This process is much faster than swapping those pages to disk drives, although it is slower than accessing real memory. Memory compression must employ lossless compression algorithms.

Data deduplication periodically calculates a unique hash number for every chunk of data by using hash algorithms such as MD5 and SHA-1. The calculated hash number is then compared against other existing hash numbers in a database that dedicates for storing chunk hash numbers. If the hash number is already in the database, the data chunk

does not need to be stored again, a pointer to the first instance is inserted in place of the duplicated data chunk. Otherwise, the new hash number is inserted into the database and the new data chunk is stored. In this way, data chunks with equal content can be merged to a single chunk and shared in a copy-on-write fashion.

In order to achieve the best effectiveness, compression and deduplication are normally applied to small data sets and big data sets, respectively. By using seven real memory traces, this paper applies deduplication to memory pages against the traditional memory compression for increasing the effective memory space. Moreover, this paper comprehensively analyzes the characteristics of memory data. Since memory is organized in pages, we take a single memory page as a basic unit of both compression and deduplication. Our experimental results demonstrate that memory deduplication significantly outperforms memory compression. Our key contributions are as follows:

(1) This paper reports that there are a large volume of pages with identical contents contained in the running memory system.
(2) This paper proposes to apply the deduplication technology to alleviate the memory bottleneck in modern computer environments.
(3) This paper analyzes the characteristics of memory data in a great detail and explores the impacts of the characteristics on the memory compression and deduplication.
(4) This paper evaluates memory deduplication against traditional memory compression and explores the performance impacts of memory deduplication.

The remainder of this paper is organized as follows. Section 2 introduces the related work. Background knowledge is presented in Section 3. Section 4 introduces the experimental environment. The characteristics of memory data are analyzed in Section 5. Sections 6 and 7 perform a comprehensive evaluation to explore the compression and deduplication behavior of memory data, respectively. Section 8 concludes the paper.

2. RELATED WORK

Memory compression has been considered as a technique to utilize memory resources more effectively. The existing technologies can be classified into two categories including full memory compression and compressed disk cache. Full memory compression keeps the entire memory compressed (with the possible exception of some specialized regions such as DMA) [11, 14]. For example, Wilson *et al.* [11] introduced compression algorithms to compress virtual memory. Their approach can adaptively determine how much memory should be compressed by keeping track of recent program behavior. The full memory compression is best illustrated by the MXT (memory extension technology) of IBM [18]. Compressed disk cache [17] employs a portion of main memory as a buffer between the main memory and the disk drive. The evicted memory pages from the regular memory are compressed and stored in the cache, thus alleviating the disk accesses. For example, Roy *et al.* [17] proposed to compress memory pages that need to be paged out and store the pages in memory, thus avoiding the large latencies of disk accesses.

However, effectively managing the compressed memory has to handle a few challenges. First, memory decompression generates significant latency that causes a critical

impact on the memory access time. This is because the compressed memory pages have to be all decompressed on the fly. Secondly, compressed memory results in variable-sized memory pages. This requires complicated design and an efficient method to maintain the mapping between the logical and the compressed address space, and reduce the memory fragmentation. Hallnor and Reinhardt [15] designed a memory hierarchy that employs a unified compression scheme encompassing the last-level on-chip cache, the off-chip memory channel, and off-chip main memory. This scheme simultaneously increases the effective on-chip cache capacity, off-chip bandwidth, and main memory size, while avoiding compression and decompression overheads between levels. Lee *et al.* [19] suggested several techniques to reduce the decompression overhead and the impact of variable-sized compressed blocks including selective compression, fixed space allocation for the compressed blocks, parallel decompression, using a decompression buffer, and so on. Tuduce and Gross [12] designed a memory compression solution that adapts the allocation of real memory between uncompressed and compressed pages and also manages fragmentation without user involvement. The method dynamically adjusts the size of memory allocation for compression based on the resource demands of each application.

Some other optimization methods are also proposed to alleviate the challenges. M.Ekman [14] proposed a main-memory compression scheme to remove decompression and translation overhead from the critical memory access path. The scheme employs a fast and simple compression algorithm by leveraging an observation that not only memory words, but also bytes, and entire blocks and pages frequently contain the value zero. X-Match [16] is a compression algorithm that is efficient at compressing small blocks of data and suitable for high-speed hardware implementation.

Recently, some research efforts are invested in using deduplication technology to alleviate the memory bottleneck in server virtualization environment, thus maximizing the number of virtual machines that can run on a physical machine of a given resource. Memory deduplication has to periodically calculate a hash value for every physical memory page. The best location to calculate hash values is in the hypervisor of a virtualized server, since only the hypervisor has a full knowledge and access privilege of all physical memory pages. Pan *et al.* [20] proposed a deferrable aggregate hypercall (DAH) mechanism to achieve both low invocation overhead and low performance impact of memory deduplication on running applications in a virtual server. Memory scanning deduplication techniques require very aggressive scan rates to identify sharing opportunities with a short life span of up to about 5 minutes. Miller *et al.* [21] proposed to use the I/O-based hints generated by read and write operations in the virtual file system to make the memory scanning process more efficient, and in consequence enable it to find and exploit short-lived sharing opportunities without raising the scan rate. In contrast to the existing works, this paper will investigate the performance behavior of memory deduplication against the traditional memory compression.

3. BACKGROUND

3.1 Data Compression

Compression relies on the fact that the data is redundant, and the redundant data follows some rules [22]. The rules can be learned and used to accurately predict the data.

By leveraging the rules, compressing a sequence of symbols is obtained by encoding the more frequent or likely symbols with shorter code words compared to the less frequent or likely symbols. Therefore, Compression can be divided into two phases including modeling and encoding that are typically interleaved with each other. Modeling detects regularities that allow a more concise representation of the information, and makes a probability distribution. Encoding is the construction of that more concise representation based on the probabilities assigned by the model [11, 23]. Many coding algorithms have been proposed. We only discuss Arithmetic algorithm, Huffman algorithm, LZ77, LZ78, LZW, and RLE, since the algorithms will be employed to compress memory data. The reader is referred to [22] for a comprehensive understanding of the algorithms.

(1) Arithmetic coding employs an interval between 0 and 1 on the real number line to represent a source. Each symbol of the source narrows this interval. The number of bits needed to represent the symbol grows with the reducing of the interval. This approach uses an explicit probabilistic model and adopts the probabilities of the source to narrow the interval. It employs an unordered list of source and their probabilities. The number line is then partitioned into subintervals on the basis of cumulative probabilities. Therefore, a high-probability symbol narrows the interval less than a low-probability symbol, so that the high probability symbol contributes fewer bits to the coded information.
(2) Huffman coding takes a list of nonnegative weights that denote the probabilities associated with the source, and constructs a full binary tree whose leaves are labeled with the weights. A set of singleton trees are constructed for each weight in the list. At each step the trees that have the two smallest weights W_i and W_j are merged into a new tree. The weight of the new tree is $W_i + W_j$, and the tree has two children represented by W_i and W_j. The weights W_i and W_j are then deleted from the list, and the weight $W_i + W_j$ is added into the list. This process continues until the weight list contains a single value.
(3) LZ77 and LZ88 are both theoretically dictionary coders. LZ77 maintains a sliding window to keep track of some amount of the most recent data. The encoder uses this data to look for matches, and the decoder employs this data to interpret the matches that the encoder refers to. This method compresses data by replacing repeated occurrences of data with references to a single copy of that data existing in the sliding window. A match is encoded by a pair of numbers (length-offset pair). LZ78 compresses data by replacing repeated occurrences of data with references to a dictionary. The dictionary is built in terms of the input data stream. LZW (Lempel–Ziv–Welch) is an improved implementation of the LZ78. LZW encodes sequences of 8-bit data as fixed-length 12-bit codes. The codes from 0 to 255 denote 1-character sequences, and the codes from 256 to 4095 are generated in a dictionary for sequences encountered in the data as it is encoded. At each phase in compression, input data are grouped into a sequence until the next character would make a sequence for which there is no code yet in the dictionary. The code for the sequence (without that character) is removed, and a new code (for the sequence with that character) is inserted to the dictionary.
(4) RLE (Run-length encoding) stores sequences in which the same data value occurs in many consecutive data elements as a single data value and count.

3.2 Data Deduplication

Data deduplication is also called intelligent compression or single-instance storage. It involves two phases including chunking and deduplication detection [24, 25]. The chunking phase splits data into non-overlapping data blocks (chunks). Each of these chunks is processed independently afterwards. The duplication detection phase detects if another chunk with exactly the same content has already been stored, by using hash algorithms such as MD5 and SHA-1. If a chunk is duplicated, the subsequent deduplicated chunks receive a pointer to the original chunk instance. This approach can effectively eliminate redundant chunks, and ensure that only the first unique instance of any chunk is actually stored. Chunking phase is very important for the quality of redundancy detection. It can be classified into four categories:

(1) Whole file chunking (WFC): WFC employs a complete file as a basis for the duplication detection. If two files are exactly the same, the first instance of the file is stored and the subsequent deduplicated files are replaced with pointers to the stored file copy. Unfortunately, the result of the change of a single bit within a file is in a totally different copy of the entire file being stored. Therefore, this method is not very effective.
(2) Fixed-size partition (FSP) [26]: FSP splits data into equal chunks that are independent of the content of the data being stored. The effectiveness of this method is highly sensitive to the sequence of data streams. For example, adding a single bit at the beginning of a file can change the boundaries and the content of all chunks in the file. This results in a failure of redundancy detection and eliminates any remaining matches, although the two file are nearly identical with only one bit shifted.
(3) Content-defined Chunking (CDC) [24]: CDC employs the data content rather than the data position within files to locate the boundaries of chunks, thus avoiding the impact of data shifting. This approach computes a hash value F for all substrings which are equal in size (usually 48 bytes) of the file. All data between two positions for that hash value F of the substring fulfills the equitation F is assigned to one chunk. The chunks have a variable size with an expected size N. Minimal and maximal chunk sizes are determined to avoid too small and too large chunks. However, the expected size N determines the granularity of duplicate elimination, thus deciding the storage utilization.
(4) Sliding Block (SB) [27]: SB divides files into fixed-size and non-overlapping chunks and calculates its signatures (4-byte MD4 along with a 2-byte rolling checksum). If two files have the same name, each chunk signature of the target file is compared against a sliding-block signature of every chunk in the source file. This method has to calculate a separate multi-byte checksum for each byte of the data. However, the checksum information for all offsets of all files is too large in contrast to the data being stored, so this approach normally performs a finer-granularity matching.

4. EXPERIMENTAL ENVIRONMENT

Wilson *et al.* [11] executed a set of real programs on an Intel x86 machine and employed Vmtrace [28] to collect traces of memory pages. The Vmtrace captures overall

amount of live data for a run of the program and writes it into a trace file. Because different program behavior influences the trace results, a specific use case is employed to generate to page images. For example, the gcc-2.7.2 compiler compiles the largest file of its own source code combine.c to generate the page images of memory [29]. The page image traces are LRU behavior sequences that contain the paging traffic for an LRU memory of some fixed size. Each record in the behavior sequences contains six fields including a compulsory tag, a fetched page number, a fetched page image, a dirtiness tag, an evicted page number, an evicted page image. Table 1 summarizes the general information of the traces.

Table 1. Features of memory page traces.

	Trace name	Description	Size (GByte)
1	espresso	A circuit simulator	1.47
2	gcc-2.7.2	A GNU C/C++ compiler	1.13
3	gnuplot	A GNU plotting utility	1.47
4	grobner	Calculated Grobner basis functions	3.28
5	lindsay	A hypercube simulator	1.11
6	p2c	A Pascal→C transformer	1.36
7	rscheme	An implementation of Scheme	0.25

Table 2. Configuration of the experimental platform.

Components	Description
CPU	Intel Core2 T6400 (2M Cache, 2.00 GHz, 800 MHz FSB)
Memory	2G, DDR2 800MHz
Hard disk	WDC WD2500BEVT-60ZCT1 (250GB /5400RPM)
Chipset	Intel 4 Series-ICH9M

In order to maintain the characteristics of the traces, we also adopted an Intel X86 machines to process the traces. Table 2 describes the configuration of our experimental platform. In order to minimize the impacts of background processes and obtain accurate results, we turned off all unnecessary processes and dedicated a partition of 100 GB for the evaluation. All the analysis in this paper is based on ASCII characters.

5. ANALYZING MEMORY DATA CHARACTERISTICS

5.1 Statistic Results

Table 3 concludes the statistic results of the page image traces. The results refer to a symbol set which has a granularity of one byte. All percentages are of the total memory data volume. The Zero column indicates that a large volume of memory data are zero bytes across seven traces. The Continuous zero column summarizes the percentage of pages that contain continuous zeros longer than 32 bytes. According to the two columns, the percentage of zeros ranges between 35.98% and 86%. This means at least one third of the memory data is zeroes. Furthermore, most of the zeros occur continuously. The

Bound column includes the percentage of memory data which is continuous zeroes longer than 32 bytes, and the continuous zeroes start or end at a page boundary. The Low column shows percentage of memory data that are low values ranging between 1 and 9. Since we use decimal to denote ASCII characters, the low values represent start of heading, start of text, end of text, end of transmission, enquiry, acknowledge, bell, backspace, and horizontal tab, respectively [30]. The Power(2, *n*) column implies the integral power-of-two values. It indicates the values of 2, 4, 8, 16, 32, 64, 128, 255 using decimal.

Entropy is normally used to measure redundancy. The entropy of a source means the average number of bits required to encode each symbol present in the source. Therefore, the compressibility grows with the decrease of the entropy value. Entropy is a useful indicator of compression ratio for a compressor. Given a set of symbols and a source in which these symbols occur, if each symbol occurs with probability, the zero-order entropy is. The Entropy column in Table 3 is calculated with zero-order entropy.

Table 3. Statistic results of the page image traces.

Trace name	Zero (%)	Continuous zeros (%)	Bound (%)	Low(%)	Power(2,*n*) (%)	Entropy
Espresso	45.94	14.61	10.34	16.25	16.46	4.19
gcc-2.7.2	56.09	29.88	6.44	11.13	12.25	3.85
gnuplot	80.72	44.53	44.50	3.29	4.66	2.01
grobner	58.34	30.91	19.78	18.62	15.35	3.30
lindsay	86.00	49.96	25.77	6.40	6.72	1.33
p2c	59.62	13.87	7.63	6.30	7.89	3.60
rscheme	35.98	10.75	6.39	17.86	17.85	4.94

5.2 Zero Distribution

Fig. 1 shows the percentage of continuous zero distribution, where X axis represents the length of continuous zero byte, and Y axis denotes the percentage. The maximum value of X axis is 4096 which is equal to one page size. Please note that the Y axes across the seven traces are in different scale, in order to have a close observation of the percentage. If we assume that the length of continuous zero is K, the number of K is N, and the size of trace is C bytes, the percentage is calculated as $(K \times N)/C$.

The compressibility is strongly related to the distribution of zero. Since the value of a tick in the X axis represents the length of continuous zero byte, the bigger the value, the more compressible the page. For example, if the length of continuous zero is 4096, this indicates that the whole page is zero. This page will achieve the highest compression ratio. Additionally, if there are more ticks concentrating in the second half of X axis, the trace will achieve higher compression ratio than that in the first half of X axis.

Fig. 1 (a) shows that three continuous zero has the highest percentage ($P3 = 0.16$, where the number 3 indicates three continuous zeros). There are some ticks distributed across X axis. For example, $P1984 = 0.02$. According to Fig. 1 (b), this trace has a larger portion of zero than that of Fig. 1 (a). However, the length of continuous zero is very short. The length normally ranges between 0 and 10. Therefore, most of the ticks concentrate in the beginning of X axis. For example, the highest percentage $P3$ is 0.07. Alt-

hough we can observe many ticks in the second half of X axis, the percentage is very low. Figs. 1 (c) and (d) demonstrate a similar pattern. Some ticks distribute in the middle of X axis (they have a relatively long length of continuous zero), and the percentages are also very high. For example, the highest percentage in Figs. 1 (c) and (d) are $P2574 = 0.12$ and $P3 = 0.15$, respectively. The distribution of ticks across Figs. 1 (e)-(g) are very much like Fig. 1 (b). Most of the long continuous zero concentrate in the beginning of X axis. It is very interesting to observe that $P4096$ of Figs. 1 (a)-(g) are 0.0056, 0.02, 0.11, 0.0006, 0.06, 0.005, and 0.09, respectively. This indicates that there are many pages full of zeroes across the seven traces.

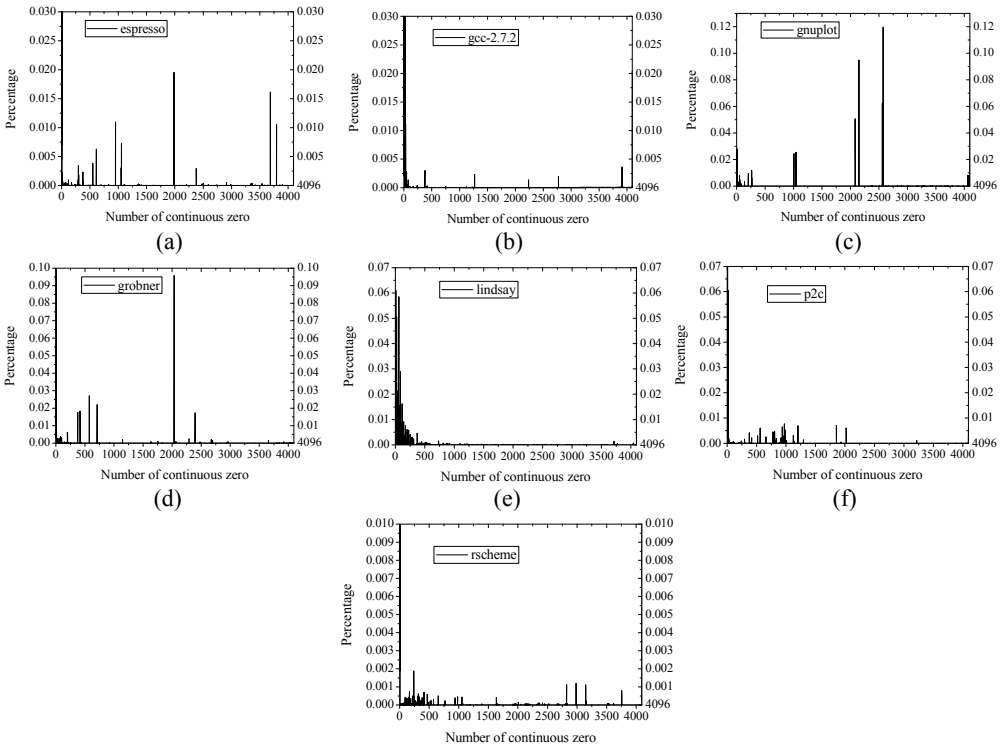


Fig. 1. Percentage of continuous zero bytes.

5.3 Symbol Distribution

Fig. 2 shows the accumulate percentage of symbol distribution across the seven traces. It is easy to observe that zeroes and low values constitute over 50% of the memory data. This is consistent with the statistics summarized in Table 3. The percentage curves grow with the increase of ASCII values. However, there is a steep growth in the very beginning of X axis, and the growth trend gradually slows. This indicates that the extended ASCII codes (The decimal of ASCII codes ranges from 128 to 255.) except 255 are only a small part of the memory data over the seven traces.

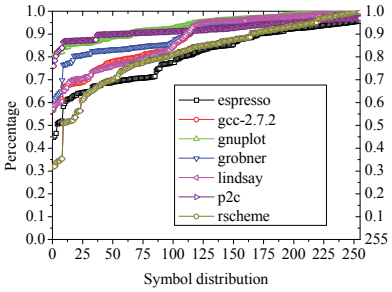


Fig. 2. Accumulate percentage of symbol distribution.

Fig. 3 deconstructs the symbol distribution across seven traces to get further insights into the distribution pattern as a complement to Fig. 2. The figures demonstrate that cluster is the most obvious feature across the seven traces. Most of the clusters are short lived at fine granularity and they appear to smooth out gradually. A number of spikes can be observed to interrupt the smoothness. Most of the spikes are accumulated within a relatively small and specific area. The spikes imply that the corresponding symbols occur frequently. The higher the spikes, the more frequent the symbols. Therefore, according to the figures, only a small number of symbols occur with a high frequency. It indicates a significant temporal locality.

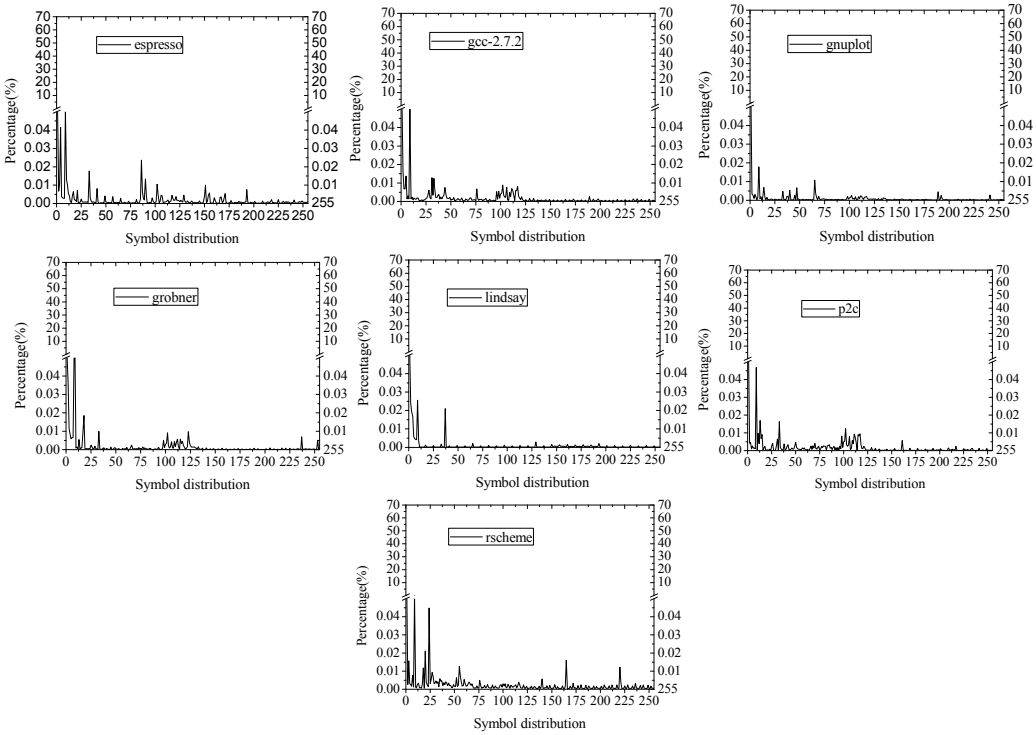


Fig. 3. Symbol distribution of seven traces.

A similar pattern across the figures is that the high clustered spikes are normally accumulated within some specific areas. For example, the beginning of X axis, around the decimal value 25, 50, 80,100, and 110. We further analyzed the collected data and found that the frequently occurred symbols normally distribute in the intervals of [0, 9], [48, 75], [65, 90], and [97, 122]. The intervals correspond to the ASCII characters summarized in Table 4.

Table 4. Some specific ASCII characters and the corresponding decimal.

Decimal Values	0→9	48→57	65→90	97→122
ASCII character	Null, start of heading, start of text, end of text, end of transmission, enquiry, acknowledge, bell, backspace, and horizontal tab	0→9	*A*→*Z*	*a*→*z*

According to the analysis, we have following conclusions: (1) Memory data contains a large portion of zeroes and the zeroes normally occur continuously; (2) A small number of symbols occur frequently. This indicates a strong temporal locality; (3) Some symbols are normally clustered and appear together. This implies a high spatial locality.

6. IDENTICAL MEMORY PAGES

In order to identify the identical memory pages contained in the traces, we calculate a hash number for the content of each memory page, and compare the hash numbers against each other. The same hash number indicates identical content of memory pages. Fig. 4 shows the percentages of identical memory pages of seven traces. It is interesting to observe that espresso, gcc-2.7.2, gnuplot, grobner, lindsay, p2c, and rscheme contain 85%, 91%, 83%, 79%, 55%, 88%, and 75% identical memory pages, respectively. The seven traces are all collected when the memory system of the computer employs LRU replacement algorithm. LRU decides which page is used least recently by checking the address of the page. Therefore, the LRU has no information about the content of the page that is replaced. This results in many identical memory pages that are associated with different addresses. For example, opening two identical files stored in different directories, or opening two similar files stored in the same directory, will incur many identical memory pages. This result can be applied to the address-based cache replacement algorithms such as LFU, SLRU, *etc.*

In order to verify the above observations and analysis, we further investigate the number of accesses going to unique page addresses and unique page contents. The traces record the information of page hit when performing the page replacement. Each page hit indicates an identical page address in the memory in contrast to the current page address. Therefore, we can calculate the number of unique address accesses. It is simple to obtain the number of unique page content accesses by using the approach employed to calculate the identical memory pages. Fig. 5 shows that the number of unique address accesses is much higher than the unique content accesses. This confirms what we report in Fig. 4.

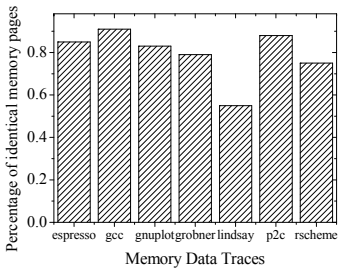


Fig. 4. Identical memory pages.

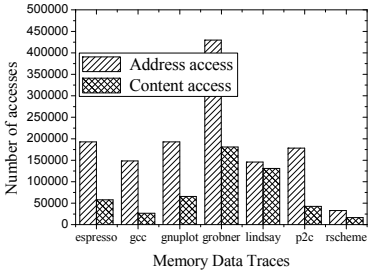


Fig. 5. Address access vs. content access.

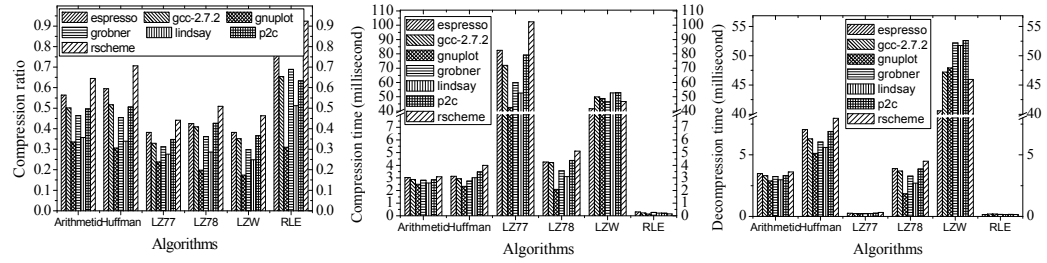
7. MEMORY DATA COMPRESSION

Lossless data compression approaches can be classified as stream compression and block compression. The streaming compression continuously accepts a stream of bytes as input and produced a compressed stream as output, while block compression accepts data block by block and compresses each block separately. Most compression methods work in the streaming mode. However, if the entire memory data is compressed as a single contiguous stream, it would be very expensive when a few memory pages are read, since the entire compressed data has to be decompressed before serving the small read requests, and this would incur some memory swapping. Therefore, it is better to compress a small group of consecutive blocks at one time. This makes the compression/decompression more efficient. When a read request comes, the system only needs to read and decompress a small group of blocks. This optimizes read operation and allows greater scalability in the total size of the memory data being compressed. Since the memory system is organized in pages, we employ block compression, and the group size is defined as an integer times of the memory page size. The compression ratio is defined as the size of compressed memory data divided by the size of uncompressed memory data.

We will quantify the benefits achieved by compressing seven memory traces. The block size is 8KB equaling to two memory pages. Fig. 6 reveals significant variations of system behavior between different compression algorithms. Fig. 6 (a) shows that the LZ algorithms (LZ77, LZ78, LZW) obtain the significant compression ratios (around 0.4), and gunplot trace achieves the best compression ratio across the six algorithms. Figs. 6 (b) and (c) demonstrate the compression and decompression time of a single block by using different algorithms and traces. It shows that the RLE algorithm performs best, and the performance of LZ77 and LZW are not acceptable in contrast to the latest Hitachi Ultrastar 15K which has an average access time of 2 milliseconds [31], since the compression time of LZ77 and decompression time of LZW are over 70 milliseconds and 50 milliseconds, respectively. According to the above discussion, we believe that LZ78 strikes a good balance between compression ratio, compression time, and decompression time. The compression time and decompression time are both less than 4 milliseconds across seven traces. Fig. 6 reveals that the compression and decompression performance difference under different algorithms can be over 500 times and 300 times, respectively.

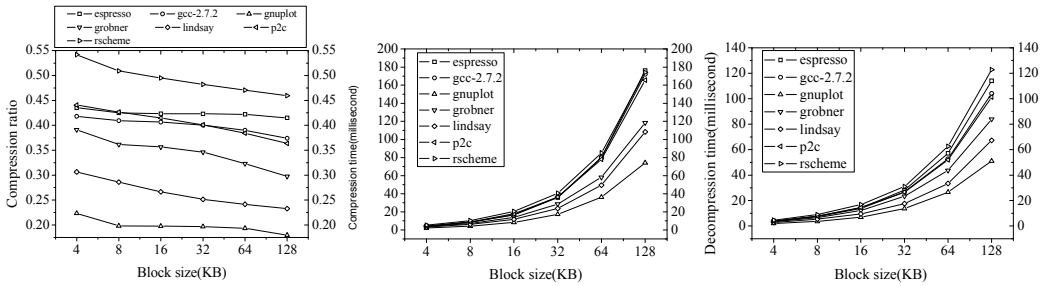
Since we use block compression to perform the evaluation, the block size has a significant impact on the system behaviour. Fig. 7 (a) shows a general trend that the com-

pression ratio decreases with the increase of block size (from 4Kbyte to 128Kbyte) across the seven traces. Figs. 7 (b) and (c) reveal that the bigger the block size is, the higher the compression and decompression time are. This pattern is reasonable, because larger data block is more compressible and requires more time to compress and decompress. However, the performance decrease is not linearly proportional to the block size.



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

Fig. 6. Impact of different algorithms on the memory data compression.

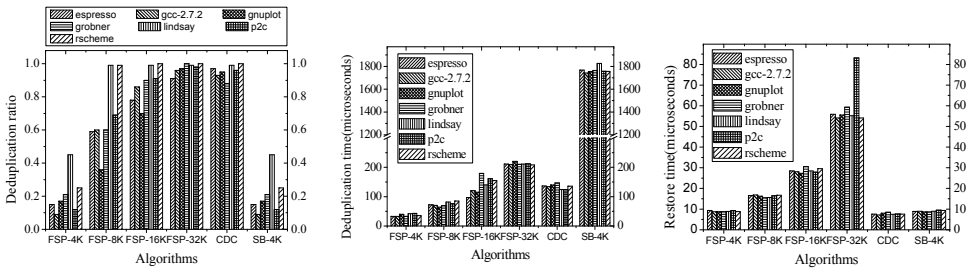


(a) Compression ratio. (b) Compression time. (c) Decompression time.

Fig. 7. Impact of block size on the system behavior (LZ78 algorithm).

8. MEMORY DATA DEDUPLICATION

Before the evaluation, we delete the header information as introduced in Section 4 in the traces and align the boundaries of chunks to the boundaries of memory pages. In contrast to the traditional compression methods, data deduplication works at a course-grained level. Fig. 8 shows the performance pattern of memory deduplication. The X axis denotes different chunking policies as discussed in Section 3.2, where the numbers following FSP and SB represent the chunking size. For example, FSP-4K implies that fixed-size partition scheme splits data into equal 4KByte chunks. Since the WFC approach is not suitable for memory deduplication, our evaluation only adopts three schemes including FSP, CDC, and SB. Please note that the Y axis of Figs. 8 (b) and (c) is in microseconds. Since the page size of the seven memory traces is 4Kbyte, FSP-4K covers one memory page, FSP-8K covers two memory pages, and so on. The deduplication ratio is defined as the size of the data deduplicated divided by the size of the original data.



(a) Deduplication ratio. (b) Deduplication time. (c) Restore time.

Fig. 8. Performance behavior of memory deduplication.

Fig. 8 (a) shows that FSP-4K and SB-4K achieve the best deduplication ratio across the seven traces. The experimental results are very close to each other by using the above two algorithms. When the chunking size of FSP is increased from 4Kbyte to 32Kbyte, the deduplication ratio is significantly increased. The compression ratio of CDC is close to 1. It is not acceptable either. Therefore, from a compression ratio standpoint, FSP-4K and SB-4K are the best candidates to perform memory deduplication. Furthermore, we believe that the optimal chunking size of memory deduplication is one memory page. Unfortunately, according to Figs. 8 (b) and (c), the deduplication time of SB-4K is about 40 times higher than that of the FSP-4K, although the restore time is comparable. According the above discussion, we believe that FSP-4K is the best candidate policy for memory deduplication, since FSP-4K takes less than 50 microseconds and 10 microseconds to deduplicate and restore a single memory page in terms of the experimental results reported in Figs. 8 (b) and (c).

According to the evaluation in Sections 6 and 7, the chunking size has an opposite impact on the performance of memory compression and memory deduplication. This is because the probability of those identical characters contained in a chunk grows with the increase of the chunk size, while the probability of two chunks that are exactly the same is decreased with the growth of the chunk size. For example, an 8Kbyte chunk would involve more compressable characters than that of a 4Kbyte. However, it is more difficult to find two identical 8Kbyte chunks than 4Kbyte chunks. Furthermore, the memory characteristics may have different impacts on the compression and deduplication. For example, gunplot trace has the best compression ratio and compression performance. However, this advantage disappears when the trace is performed deduplication.

9. COMPARISON OF MEMORY COMPRESSION AND DEDUPLICATION

In order to further investigate the performance behaviour of memory deduplication, we compare the memory compression against memory deduplication in terms of time overhead and data reduction ratio. As discussed in Sections 7 and 8, LZ78 algorithm and FSP-4K achieve the best performance in the compression and deduplication, respectively. Therefore, we only compare the performance behaviour of LZ78 against FSP-4K. Fig. 9 (a) shows the compression time against deduplication time. It demonstrates that the deduplication time is much faster than that of compression time across seven traces. Fig. 9 (b) depicts the decompression time against restore time. It shows a similar trend. It

indicates that the restore time significantly outperforms the decompression time when using the seven traces. For example, it takes only 8.83 microseconds to restore the rscheme trace data when employing FSP-4K. However, the decompression time of LZ78 grows to 4620 microseconds. Fig. 9 (c) demonstrates the compression ratio against deduplication ratio when the seven traces are adopted to perform the evaluation. It shows that the deduplication ratio is much better than that of compression ratio. This means that there are many identical pages contained in the traces. These pages generate the high deduplication ratio. However, this characteristic cannot be leveraged by compression, because compression is only performed within every single memory page.

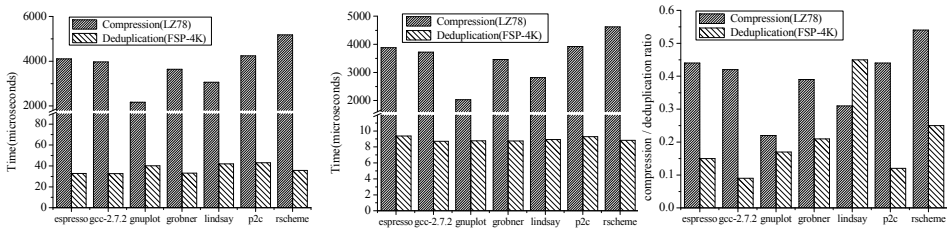


Fig. 9. Comparison of memory compression and deduplication.

10. CONCLUSION

This paper explores the performance behavior of memory deduplication against memory compression by using seven real memory traces. The experimental results give the following indications:

(1) There is a large volume of memory pages with identical contents contained in the memory system, and the number of unique memory content accessed is much less than the unique memory address accessed. Therefore, memory deduplication significantly outperforms memory compression.
(2) FSP achieves the best performance in contrast to CDC and SB when performing memory deduplication. The optimal chunking size of FSP is equal to the size of a memory page. A specific memory data that is very compressable may not be able to achieve good deduplication performance. The characteristics of memory data have different impacts on compression and deduplication.

The analysis results in this paper should be able to provide useful insights for designing or implementing systems that require abundant memory resources.

ACKNOWLEDGMENT

This work is supported by the NSFC (61572232, 61272073), Science and Technology Planning Project of Guangzhou (201604016100), NSF of Guangdong Province (S2013020012865), and Open Research Fund of Key Laboratory of Computer System

and Architecture, Institute of Computing Technology, Chinese Academy of Sciences (CARCH201401).

REFERENCES

1. Y. Deng, "Exploiting the performance gains of modern disk drives by enhancing data locality," *Information Sciences*, Vol. 179, 2009, pp. 2494-2511.
2. Y. Deng, "What is the future of disk drives, death or rebirth?" *ACM Computing Surveys*, Vol. 43, 2011, Article 23.
3. Hitachi Global Storage Technologies – HDD Technology Overview Charts, http://www.hitachigst.com/hdd/technolo/overview/storagetechchart.html.
4. W. W. Hsu and A. J. Smith, "The performance impact of I/O optimizations and disk improvements," *IBM Journal of Research and Development*, Vol. 48, 2004, pp. 255-289.
5. S. Schlosser, J. Griffin, *et al.*, "Designing computer systems with MEMS-based storage," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 1-12.
6. Y. Deng, F. Wang, and N. Helian, "EED: energy efficient disk drive architecture," *Information Sciences*, Vol. 178, 2008, pp. 4403-4417.
7. Y. Deng, J. Cai, W. Jiang, and X. Qin, "Employing dual-block correlations to reduce the energy consumption of disk drives," *Computing*, Vol. 99, 2017, pp. 235-253.
8. K. Zhou, Y. Liu, *et al.*, "Deep self-taught hashing for image retrieval," in *Proceedings of the 23rd ACM International Conference on Multimedia*, 2015, pp. 1215-1218.
9. Z. Huang, H. Jiang, *et al.*, "XI-Code: A family of practical lowest density MDS array codes of distance 4," *IEEE Transactions on Communications*, Vol. 64, 2016, pp. 2707-2718.
10. J. M. Rodriguez, C. Mateos, *et al.*, "Energy-efficient job stealing for CPU-intensive processing in mobile devices," *Computing*, Vol. 96, 2014, pp. 87-117.
11. P. R. Wilson, S. F. Kaplan, *et al.*, "The case for compressed caching in virtual memory systems," in *Proceedings of Annual Conference on USENIX ATC*, 1999, pp. 1-16.
12. I. Tuduce and T. Gross, "Adaptive main memory compression," in *Proceedings of Annual Conference on USENIX ATC*, 2005, pp. 237-250.
13. Y. Zhao, H. Jiang, *et al.*, "DREAM-(L)G: A distributed grouping-based algorithm for resource assignment for bandwidth-intensive applications in the cloud," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 27, 2016, pp. 3469-3484.
14. M. Ekman and P. Stenstrom, "A robust main-memory compression scheme," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005, pp. 1-12.
15. E. G. Hallnor and S. K. Reinhardt, "A unified compressed memory hierarchy," in *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005, pp. 1-12.
16. M. Kjelsg, M. Gooch, and S. Jones, "Design and performance of a main memory hardware data compressor," in *Proceedings of the 22nd Euromicro Conference*, 1996, pp. 423-430.

17. S. Roy, R. Kumar, and M. Prvulovic, "Improving system performance with compressed memory," in *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, 2001, pp. 1-7.
18. B. Abali, H. Franke, S. Xiaowei, *et al.*, "Performance of hardware compressed main memory," in *Proceedings of the 7th International Symposium on H High-Performance Computer Architecture*, 2001.
19. J. Lee, W. Hong, and S. Kim, "Design and evaluation of a selective compressed memory system," in *Proceedings of International Conference on Computer Design*, 1999, pp. 1-8.
20. Y. Pan, J. Chiang, *et al.*, "Hypervisor support for efficient memory de-duplication," in *Proceedings of the 17th IEEE International Conference on Parallel and Distributed Systems*, 2011, pp. 33-39.
21. K. Miller, *et al.*, "KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient," in *Proceedings of RESoLVE12*, 2012, pp. 1-12.
22. D. A. Lelewer and D. S. Hirschberg, "Data compression," *ACM Computing Surveys*, Vol. 19, 1987, pp. 261-296.
23. M. Nelson, *et al.*, *The Data Compression Book*, M&T Books, NY, 1995.
24. D. Meister and A. Brinkmann, "Multi-Level comparison of data deduplication in a backup scenario," in *Proceedings of the Israeli Experimental Systems Conference*, 2009, pp. 1-12.
25. B. Zhu, K. Li, and H. Patterson. "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the 6th USENIX FAST*, 2008, pp. 269-282.
26. D. Bobbarjung, S. Jagannathan, and C. Dubnicki, "Improving duplicate elimination in storage systems," *ACM Transactions on Storage*, Vol. 2, 2006, pp. 424-448.
27. N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replica synchronization," in *Proceedings of the 4th USENIX FAST*, 2005, pp. 281-294.
28. VMTrace, http://linux-mm.org/VmTrace.
29. P. Wilson, M. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: a survey and critical review," in *Proceedings of International Workshop on Memory Management*, 1995, pp. 1-78.
30. Ascii table, http://www.asciitable.com/.
31. Ultrastar 15K147, 2009, "Ultrastar 15K147 hard disk drives specifications," http://www.hitachigst.com/hdd/support/15k147/15k147.htm.

YuHui Deng is a Professor at the Computer Science Department of Jinan University. Before joining Jinan University, he worked at EMC Corporation as a senior research scientist from 2008 to 2009. He worked as a research officer at Cranfield University in the United Kingdom from 2005 to 2008. He has authored and coauthored more than 60 refereed papers. His research interests cover information storage, cloud computing, green computing, computer architecture, *etc.*

XinYu Huang was a master student at the Data Storage and Cluster Computing Lab, Computer Science Department of Jinan University. He received his ME degree from Jinan University in 2012. His research interests include information storage and system security.

LiangShan Song was a master student at the Data Storage and Cluster Computing Lab, Computer Science Department of Jinan University. He received his ME degree from Jinan University in 2013. He is currently a software engineer at Tencent. His research interests include information storage and data deduplication.

YongTao Zhou was a master student at the Data Storage and Cluster Computing Lab, Computer Science Department of Jinan University. He received his ME degree from Jinan University in 2016. He is currently a software engineer at Tencent Holdings Ltd. His research interests include data storage, data deduplication, file system, and cloud.

Frank Wang is Head of School, School of Computing, University of Kent, UK. He was the director of Centre for Grid Computing, Cambridge-Cranfield High Performance Computing Facility (CCHPCF), Cranfield University. He was Chair in e-Science and Grid Computing. He is on the High End Computing Panel for the Science Foundation Ireland (SFI). He was the Chair (UK & Republic of Ireland Chapter) of the IEEE Computer Society.