LZ4m: A Fast Compression Algorithm for In-Memory Data

Se-Jun Kwon*, Sang-Hoon Kim†, Hyeong-Jun Kim*, and Jin-Soo Kim*
*College of Information and Communication Engineering, Sungkyunkwan University
Email: sejun000@csl.skku.edu, hjkim@csl.skku.edu, jinsookim@skku.edu
†School of Computing, Korea Advanced Institute of Science and Technology (KAIST)
Email: sanghoon@calab.kaist.ac.kr

***Abstract*—Compressing in-memory data is a cost-effective solution for dealing with the memory demand from data-intensive applications. This paper proposes a fast data compression algorithm for in-memory data that improves performance by utilizing the characteristics frequently observed from in-memory data.**

I. Introduction

Dealing with the ever-increasing memory requirements of data-intensive applications is the subject of intensive work in both industry and academia. In particular, it becomes an important issue in consumer devices as their fast evolution places a high demand on memory. As a cost-effective solution for the demand, manufacturers attempt to compress in-memory data thereby increasing the effective memory size and lowering manufacturing cost [1], [2]. Specifically, in-memory data compression techniques become important since the market for consumer devices becomes extremely price sensitive and sales are shifted from premium to cheaper models equipped with small memory [3].

Data compression, or compression shortly, refers to the technique that reduces the size of data by representing the data in a more concise form. There have been many studies that employ the compression for in-memory data, ranging from computer architecture systems to operating systems. For instance, some memory technologies [4], [**?**] compress cache lines at the memory controller before writing them to memory, providing a larger amount of memory than physically equipped one. Compcache and zram [5] find out rarely referenced page frames and compress them to reduce their memory footprint.

A compression algorithm for in-memory data must be fast in both compression and decompression while providing acceptable compression ratio[1] as the performance of memory access heavily influences on overall system performance. To this end, Wilson et al. have proposed the WKdm algorithm that utilizes the regularities that are frequently observed from in-memory data [6]. However, although WKdm exhibits outstanding compression speed, its poor compression ratio is the major concern that hinders the application of the algorithm. On the other hand, general-purpose compression algorithms such as Deflate [7], LZ4 [8], and LZO [9] compress in-memory data better than WKdm. However, they exhibit poor performance in terms of compression and decompression speed, thereby impairing system performance.

This paper proposes a new novel compression algorithm called LZ4m, which stands for **LZ4** for in-**m**emory data. We expedite the scanning of input stream of the original LZ4 algorithm by utilizing the characteristics frequently observed from in-memory data. We also revise the encoding scheme of the LZ4 algorithm so that the compressed data can be represented in a more concise form. Our evaluation results with the data obtained from a running mobile device show that LZ4m outperforms previous compression algorithms in compression and decompression speed by up to 2.1× and 1.8×, respectively, with a marginal loss of less than 3% in compression ratio.

II. Ziv-Lempel Algorithm and LZ4

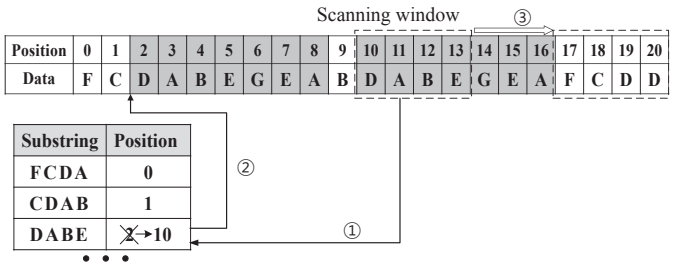
Many compression algorithms aiming at low compression latency are based on the *Lempel-Ziv algorithm* [10]. The key strategy of Lempel-Ziv algorithm is to scan an input stream and find the *longest* match between the substring starting from the current scanning position and the substrings in the already scanned part of the input stream. If such a match is found, the matched substring at the current position is substituted with a pair of the backward offset to the match and the length of the match. However, identifying the *longest* match can incur high overheads in time and space. Thus, algorithms in practice usually alter the strategy to quickly identify a substring that is *long enough* to be compressed [7], [8], [9].

LZ4 is one of the most popular and successful compression algorithms based on the Lempel-Ziv algorithm. Figure 1 illustrates how LZ4 identifies the match and encodes an input stream. LZ4 scans an input stream with a 4-byte window ("DABE" at position 10 in this case) and checks whether the substring in the window appeared before. To assist the match, LZ4 maintains a hash table that maps a 4-byte substring to a position from the start of the input stream. If the hash table contains an entry for the current 4-byte substring in the scanning window (Figure 1(a)-①), it implies that the current substring appeared at the certain position in the previously scanned stream (Figure 1(a)-②). Therefore, the substring starting at the position has an identical prefix with

This work was supported by the National Research Foundation of Korea (NRF) grant founded by the Korea Government (MSIP) (No. NRF-2016R1A2A1A05005494).

[1]Throughout the paper, we use the term *compression ratio* to refer to the ratio of the compressed size to the original size.

978-1-5090-5544-9/17/$31.00 ©2017 IEEE

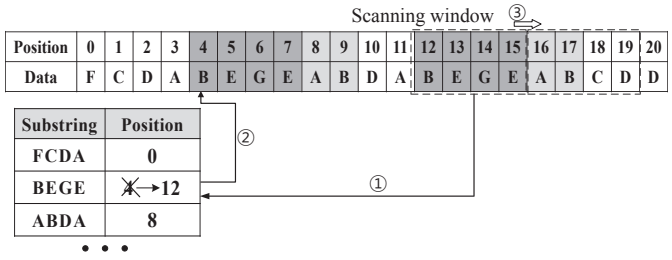


(a) Matching process of LZ4

Token		Body			
Literal length	Match length	Literal		Match	
		Length	Data	Offset	Length
4 bits	4 bits	0+ bytes	0+ bytes	2 bytes	0+ bytes

(b) Encoding of LZ4

Fig. 1. Matching process and encoding of LZ4



(a) Matching process of LZ4m

Token			Body			
Offset	Literal length	Match length	Literal		Match	
			Length	Data	Offset	Length
2 bits	3 bits	3 bits	0+ bytes	0+ bytes	1 byte	0+ bytes

(b) Encoding of LZ4m

Fig. 2. Matching process and encoding of LZ4m

the substring starting at the current scanning position. LZ4 utilizes this property and finds the longest match starting from these two positions. In this example, the longest match string is "DABEGEA". Then, the corresponding hash table entry is updated to the current scanning position and the scanning window is slid forward by the length of the match (Figure 1(a)-③). If no entry for the current substring is found in the hash table, a new entry is inserted into the hash table and the scanning window is advanced by one byte. The scanning is repeated until the scanning window reaches the end of the stream.

According to the matching scheme, the input stream is partitioned into a number of substrings, called *literals* and *matches*. The literal means a substring that does not match any of previously appeared substrings. The literal is always followed by one or more matches unless the literal is the last substring of the input stream. LZ4 encodes the substrings into *encoding units* which are comprised of a *token* and a *body* as shown in Figure 1(b). Every encoding unit is started with a 1-byte token. If the substring to encode is a literal followed by a match, the upper 4 bits of the token are set to the length of the literal and the lower 4 bits are set to the length of the following match. If 4 bits are not enough to represent the length, i.e., the substring is longer than 15 bytes, the corresponding bits are set to $1111_{(2)}$, and the length subtracted by 15 is placed on the body which follows the token. The body contains the literal data and the description of the match, which is comprised of a backward offset to the match and the length of the match. As the offset is encoded in 2 bytes, LZ4 can look back up to 64 KB to find a match. Additional matches immediately followed by a certain match are encoded in the similar way, except that the literal length field in the token is set to $0000_{(2)}$ and the body omits the part for the literal.

III. Proposed LZ4m algorithm

LZ4 focuses on the compression and decompression speed with an acceptable compression ratio. However, as LZ4 is designed as a general-purpose compression algorithm, it does not utilize the inherent characteristics of in-memory data.

In-memory data consists of virtual memory pages that contains the data from the stack and the heap regions of applications. The stack and the heap regions contain constants, variables, pointers, and arrays of basic types which are usually structured and aligned to the word boundary of the system [6]. Based on the observation, we can expect that data compression can be accelerated without a significant loss of compressing opportunity by scanning and matching data in the word granularity. In addition, as the larger granularity requires less bits to represent the length and the offset, the substrings (i.e., literals and matches) can be encoded in a more concise form.

We revise the LZ4 algorithm to utilize these characteristics and propose the *LZ4m* algorithm which stands for LZ4 for in-memory data. Figure 2(a) illustrates the modified compression scheme and the unit encoding of LZ4m. LZ4m uses the same scanning window and hash table of the original LZ4. In contrast to the original LZ4 algorithm, LZ4m scans an input stream and finds the match in a 4-byte granularity. If the hash table indicates no prefix match exists, LZ4m advances the window by 4 bytes and repeats identifying the prefix match. As shown in the Figure 2(a), after examining the prefix at position 8, the next position of the window is 12 instead of 9. If a prefix match is found, LZ4m compares subsequent data and finds the longest match in the 4-byte granularity as well. In the example, although there is a 6-byte match starting from position 12, LZ4m only considers the 4-byte match from 12 to 15, and slides the scanning window forward by four bytes, to position 16 (Figure 2(a)-③).

We also optimize the encoding scheme to utilize the 4-byte granularity. As the offset is aligned to the 4-byte boundary and the length is a multiple of 4 bytes, two least significant bits of

978-1-5090-5544-9/17/$31.00 ©2017 IEEE

these fields are always $00_{(2)}$. Thus, every field representing the length and the offset (the literal length and the match length in the token, and the literal length, the match length, and the match offset in the body) can be shortened by 2 bits. Moreover, as LZ4m is targeting to compress 4 KB pages in a 4-byte granularity, the field for the match offset requires at most 10 bits. Consequently, we place the two most significant bits of the match offset in the token and put the remaining 8 bits in the body.

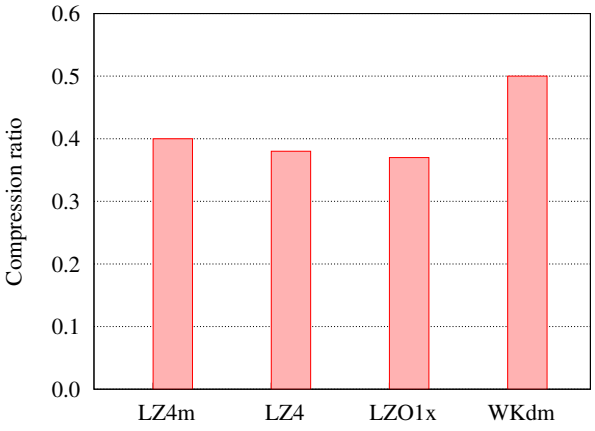
IV. Evaluation

We evaluated the proposed LZ4m algorithm using real in-memory data obtained from a mobile device. We evaluated LZ4 and LZO1x as the representatives of general-purpose algorithms, and WKdm as a specialized one. To collect in-memory data, particularly the one that in-memory data compression schemes are interested in, we collected data that are evicted from main memory via swapping. We set up a mobile system platform on a mobile system development board. And then we configured the kernel to swap data on a custom block device emulated with a SSD development platform board. The custom block device collects swap data to a particular partition that can be examined later off-line. We collected 98,995 pages (386.7 MB) of swap data while launching 24 popular applications 512 times in total.

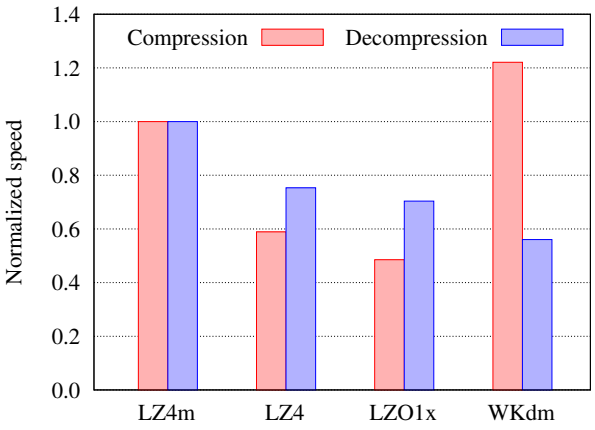
Figure 3 compares the algorithms in terms of their compression ratio and compression/decompression speed. Recall that the compression ratio is the average of pages, and the smaller compression ratio implies the smaller compressed size for the same data. Also, the speeds are normalized to that of LZ4m. Compared to the general-purpose algorithms (i.e., LZ4 and LZO1x), LZ4m shows the comparable compression ratio which is only degraded by up to 3%. However, LZ4m outperforms these algorithms in speed by up to 2.1× and 1.8× for compression and decompression, respectively. On the other hand, LZ4m outperforms WKdm significantly in compression ratio and decompression speed at the cost of 21% slowdown in compression speed. These results suggest that LZ4m substantially improves the compression/decompression speed of LZ4 with a marginal loss of compression ratio.

Figure 4 shows a cumulative distribution of compression ratio for pages. The compression ratio curve of LZ4m does not fall much behind those of the LZO and LZ4 algorithms. However, WKdm shows distinctively compression ratio curve which lags much behind other algorithms. In addition, 6.8% of pages cannot be compressed at all with WKdm whereas less than 1% does with others. This suggests that the WKdm's speed-up in compression can be offset by its poor compression ratio.

To further analyze the implication of the 4-byte granularity of the match offset and the match length, we counted the length of match substrings from the trace. Figure 5 compares the results from the original LZ4 and LZ4m with a cumulative count of matches over match length. The inset magnifies the original result whose match lengthes are between 0 to 32. We can verify that the increased granularity decreases the total



(a) Compression ratio



(b) Compression/decompression speed

Fig. 3. Compression ratio and compression/decompression speed of various compression algorithms

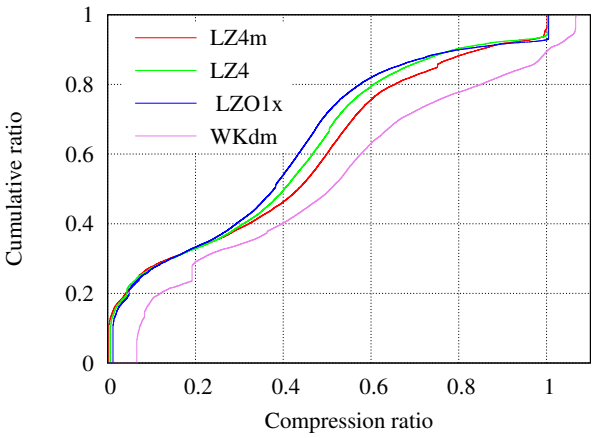


Fig. 4. Cumulative distibution of compression ratio

occurrence of matches only by 2.5%, which implies that the 4-byte granularity scheme influences just little on the opportunity to find a match. Also, the distribution of the match lengths shows that an n-byte match in the original scheme can be

978-1-5090-5544-9/17/$31.00 ©2017 IEEE

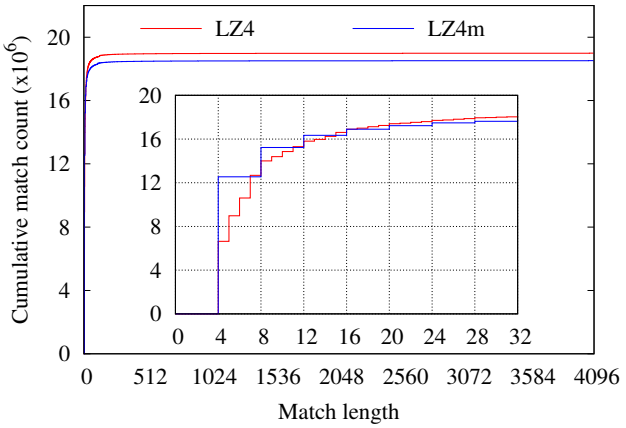


Fig. 5. Cumulative count of matches over match length

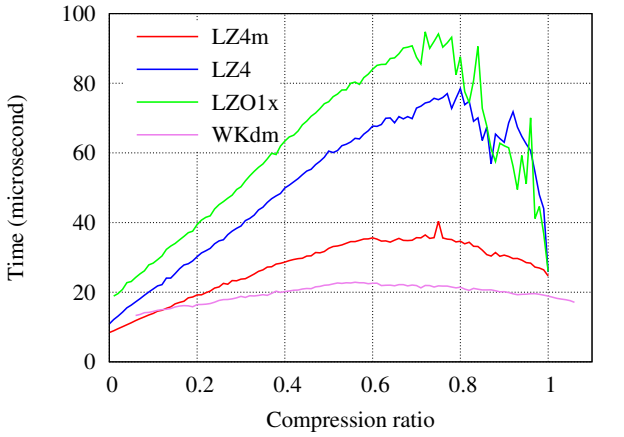


Fig. 6. Average compression speed over compression ratio

substituted with a $\lfloor n/4 \rfloor \times 4$-byte match. This suggests that the disadvantage of the 4-byte granularity in the match length is also marginal.

Figure 6 shows the relationship between the compression speed and the compression ratio of the algorithms. The speed is obtained by measuring the time to compress each page and averaging the time of pages having the same compression ratio. The time to compress well-compressed pages (pages with small compression ratio values) are similar across the algorithms. For badly-compressed pages (pages with large compression ratio values), however, LZ4m shows outstanding compression speed compared to LZ4 and LZO1x. This considerable amount of speedup is originated from the scanning process of LZ4m that advances the scanning window by 4 bytes if no prefix match is found, expediting the scanning by four times on the hardly-compressible pages.

Figure 7 shows the relationship between the decompression speed and the compression ratio of the algorithms. The speed is obtained in the same way as the average compression speed. We can confirm that LZ4m outperforms other algorithms in decompression speed over almost entire range of the compression

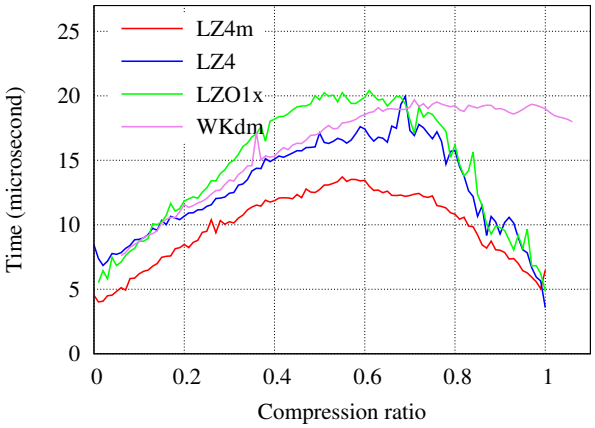


Fig. 7. Average decompression speed over compression ratio

ratio. Interestingly, WKdm does not excel other algorithms in decompression speed, especially for the pages that are not compressed well, although it promises outstanding compression speed. This is a critical drawback as a compression algorithm for in-memory data since decompression usually happens on performance-critical paths in memory subsystems, such as fault handling or swap-in.

V. Conclusion

We optimized a popular general-purpose compression algorithm by utilizing the inherent characteristics of in-memory data. Evaluation with real-life data confirms that the proposed LZ4m greatly boost the compression/decompression speed without substantial loss in compression ratio. We plan to apply the LZ4m to a real in-memory compression system and to measure its implication on the overall system performance.

References

[1] Samsung Electronics, Co., "Galaxy Note 3." [Online]. Available: http://www.samsung.com/uk/consumer/mobile-devices/smartphones/galaxy-note/SM-N9005ZKEBTU
[2] Android. Low RAM. [Online]. Available: https://source.android.com/devices/tech/low-ram.html
[3] International Data Corporation (IDC), "Worldwide quarterly mobile phone tracker," Aug. 2015. [Online]. Available: http://www.idc.com/tracker/showtrackerhome.jsp
[4] S. Arramreddy, D. Har, K.-K. Mak, R. B. Tremaine, and M. Wazlowski, "IBM "MXT" memory compression technology debuts in a serverworks northbridge," in *Proceedings of Hot Chips 12*, 2000.
[5] F. Douglis, "The compression cache: Using on-line compression to extend physical memory," in *Proceedings of the Winter USENIX Conference*, 1993, pp. 519–529.
[6] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis, "The case for compressed caching in virtual memory system," in *Proceedings of the 1999 USENIX Annual Technical Conference*, June 1999, pp. 101–116.
[7] J. loup Gailly and M. Adler. (2013) zlib:a massively spiffy yet delicately unobtrusive compression library. [Online]. Available: http://www.zlib.net/
[8] Y. Collect. (2013) LZ4: extremely fast compression algorithm. [Online]. Available: http://code.google.com/p/lz4
[9] M. F. Oberhumer. (2011) LZO real-time data compression library. [Online]. Available: http://www.oberhumer.com/opensource/lzo
[10] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on Information Theory*, vol. 23, 1977.

978-1-5090-5544-9/17/$31.00 ©2017 IEEE