Energy Efficient and Low-Cost Server Architecture for Hadoop Storage Appliance

Do Young Choi, Jung Hwan Oh, Ji Kwang Kim, and Seung Eun Lee*
Department of Electronic Engineering
Seoul National University of Science and Technology
Seoul, Korea
[e-mail: {choidoyoung, ohjunghwan, jikwang.kim, seung.lee}@seoultech.ac.kr]
*Corresponding author: Seung Eun Lee

Received June 10, 2020; revised September 16, 2020; accepted November 28, 2020; published December 31, 2020

Abstract

This paper proposes the Lempel-Ziv 4(LZ4) compression accelerator optimized for scale-out servers in data centers. In order to reduce CPU loads caused by compression, we propose an accelerator solution and implement the accelerator on an Field Programmable Gate Array(FPGA) as heterogeneous computing. The LZ4 compression hardware accelerator is a fully pipelined architecture and applies 16 dictionaries to enhance the parallelism for high throughput compressor. Our hardware accelerator is based on the 20-stage pipeline and dictionary architecture, highly customized to LZ4 compression algorithm and parallel hardware implementation. Proposing dictionary architecture allows achieving high throughput by comparing input sequences in multiple dictionaries simultaneously compared to a single dictionary. The experimental results provide the high throughput with intensively optimized in the FPGA. Additionally, we compare our implementation to CPU implementation results of LZ4 to provide insights on FPGA-based data centers. The proposed accelerator achieves the compression throughput of 639MB/s with fine parallelism to be deployed into scale-out servers. This approach enables the low power Intel Atom processor to realize the Hadoop storage along with the compression accelerator.

Keywords: Hadoop Storage, Hardware Accelerator, Lempel-Ziv 4 Algorithm, Data Compression

This work is supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT(No. 2019R1F1A1060044).

http://doi.org/10.3837/tiis.2020.12.002 **ISSN : 1976-7277**

1. Introduction

Big data processing is getting more important since the advent of technologies such as big data services, cloud computing, and internet of things (IoT). Distributed computing is widely used to process the enormous amounts of data, and the Hadoop is an open source framework that supporting the distributed computing [1, 2]. Based on the scale-out architecture, the Hadoop processes an enormous amounts of data by clustering numerous servers [3]. The architecture of Hadoop enables the cluster performance to increase linearly by extending the number of servers on the distributed computing framework. Also, the distributed computing based Hadoop system has advantages of data accessibility and security. Nevertheless, adopting a high-performance CPU to each server causes total cost of ownership (TCO) and power consumption issues. For those reasons, several studies were researched to address cost and power issues while improving Hadoop's data processing capabilities. A recent study adopted the compression hardware accelerator to each server to lessen the burden of CPU [4, 5]. As Hadoop provides Bzip2, Zlib, LZO, Zstandard, Snappy, and LZ4 (Lempel-Ziv 4) compression algorithms [6], these algorithms can be adopted to the compression hardware accelerator to lessen the workload of compression operation. By encoding the original data, the storage capacity and network bandwidth can be improved.

The capacity of Hadoop workload increases linearly as adopting new servers on the cluster. However, due to the TCO and power issues, there is a limitation of increasing the number of servers in the Hadoop framework [7, 8]. Although replacing the high-performance processor with a low-performance processor address the server expansion limitation, this alternative causes bandwidth and storage performance degradation. Therefore, it is necessary to offload the burden of compression operations of low performance processor. In this paper, based on the possibility of LZ4 hardware accelerator, we propose to enhance the performance of hadoop storage appliance by adopting the low performance processors instead of the high-performance processor. By using the Intel Atom processor [9] on the Hadoop server, the power and cost issues can be solved. Furthermore, compression throughput can be improved by using the LZ4 hardware accelerator. In addition, offloading the workload to the compression hardware accelerator can enhance the transmission bandwidth and storage space by compressing the massive amounts of data with high compression throughput.

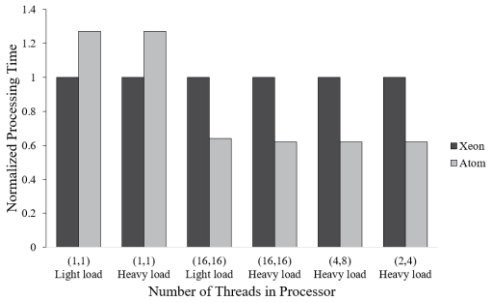
When implementing the compression hardware accelerator on Hadoop storage server, the LZ4 is an suitable algorithm compared to other compression algorithms. However, the match process in LZ4 algorithm still needed to be optimized for hardware implementation. Realizing LZ4 hash table as software has significantly less limitation in the size of buckets because there is less memory limitation compared to the hardware implementation. However, implementing the hash table as hardware has the disadvantage of increasing the hardware resource derived from the complex hash function and bucket. In addition, there is a compression throughput issue when the compression hardware accelerator uses only one dictionary, because the match procedure have to be performed several times in one match procedure. To address these issues, the LZ4 hardware accelerator exploits an ASCII address based dictionary which reduces the complexity of hash function. Also, the compression throughput of LZ4 hardware accelerator can be enhanced by adopting multiple dictionaries in parallel as a way to reduce the overhead of the match procedure. The contributions of our work in this paper are as follows.

- Improve the compression throughput and reduce the hardware resource of LZ4 hardware accelerator as customizing the LZ4 compression to hardware implementation.
- Replace the high-performance processors used for servers with the low-performance processors and improve the energy, and cost performance of Hadoop framework.
- Offload the burden of compression operations caused by processor replacement to LZ4 hardware accelerator and address the server extension limitation of Hadoop framework.

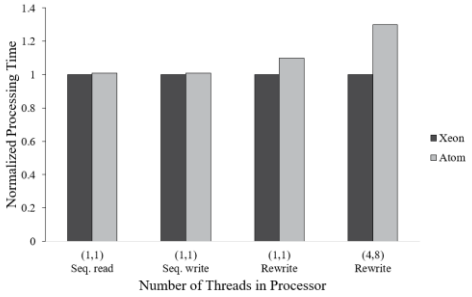
The structure of this paper is organized as follows. Section 2 presents the motivation with the results of comparison between Intel Atom processor [9] and Intel Xeon processor [10] to verify the possibilities of adopting Atom processor on Hadoop servers. Section 3 provides the related work about LZ4 acceleration. Section 4 explains compression algorithms of Hadoop, LZ4 algorithm, frame format, and dictionary. Section 5 proposes micro-architecture of LZ4 hardware accelerator and the parallel dictionary optimized to the hardware accelerator. Section 6 provides performance analysis of LZ4 hardware accelerator with high compression throughput compared to software-based LZ4. Finally, section 7 concludes the paper.

2. Motivation

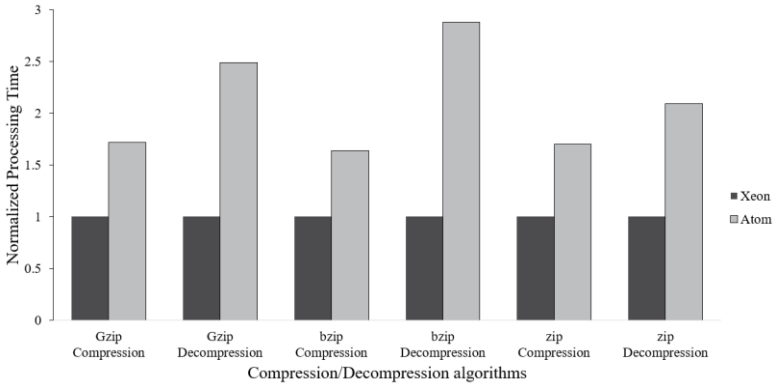
For large size of web applications, frequent data I/O and random access of huge datasets occurs in the clusters, but the basic arithmetic operations are hardly peformed. A half of the TCO of clusters with these workloads is due to power consumption. From 55 to 60 % of power consumption derives from CPU and memory. Thus, a server with a low-power CPU-based architecture would be the good option for the clusters with these workloads. Prior to implement a scale-out server for Hadoop distributed file system, we evaluated the performance of a high-performance Intel Xeon processor (E5-2609, 2.4 GHz, 4 cores) and low-power Intel Atom processor (C2758, 2.4Ghz, 8 cores) by performing major functions such as sequential read/write, compression/decompression and file I/O. **Fig. 1** shows the performance analysis results to inspect the Xeon and Atom processor. In **Fig. 1**(a), the processor test repeats the 64bit integer calculations. (A, B) implies that A is the number of threads in Xeon processor and B is the number of threads in Atom processor. Y-axis implies the normalized processing time in each analysis. Also, there are light and heavy workloads to compare the performance by differentiating the number of threads in processors. In the case of single thread with both workloads, the Xeon is faster than Atom as the Xeon aims to handle large computational capacities. However, the Atom processor shows the better performance in multi-threads. File I/O speed is the major factor to enhance the overall performance of Hadoop clusters. In **Fig. 1**(b), the analysis result of file I/O speed is quite similar in sequential write and read process in single thread. In the case of rewrite process, both processors achieves the acceptable performance with a single thread. However, the Xeon is much faster thanks to multithreading. In **Fig. 1**(c), we test compression and decompression speed of the three compression algorithms, which are exploited in Hadoop including Gzip, Bzip, and Zip. Atom-based server is 1.6× slower in compression and 2.5× slower in decompression than Xeon-based server. This result is because the Xeon processor is highly specific to the successive computational workloads like compression and decompression.



(a) 64bit integer calculation speed



(b) File I/O speed



(c) Compression/Decompression speed

Fig. 1. Performance analysis of Xeon VS. Atom processor

In Hadoop system, the integer calculations are not the main workloads. The Hadoop framework performs data compression and decompression while data I/O procedure to utilize the network bandwidth and save storage. Therefore, the compression and decompression are the most influential factors in to the overall system performance.

According to the processor performance analysis, the usage of Atom processor on Hadoop storage server guarantees only in terms of the file I/O performance compared to the conventional Xeon-based server. However, the performance degradation is expected due to the frequent data compression and decompression whenever the data I/O occurs. Based on the analysis, we figure out that the Atom-based micro-server requires the acceleration to catch up with the computation capacities of Xeon-based servers. The offloading is beneficial with large amounts of computation so that we focus on the compression procedure in Hadoop cluster, which is the main factor of performance degradation. Therefore, we propose the hardware accelerator to offload heavy data compression in Hadoop clusters. The Atom-based server can guarantee the low cost to operate and maintain the storage services.

3. Related Work

Among the compression algorithms for Hadoop framework, the LZ4 has the highest compression throughput with a fair compression ratio. And LZ4 is a suitable compression algorithm for hardware implementation compared to other compression algorithms. Therefore, Hadoop compression hardware accelerator has advantages of compression throughput and hardware implementation when LZ4 compression algorithms are realized on the hardware accelerator.

LZ4 created by Yan Collet is a lossy, dictionary-based, and byte-oriented compression algorithm [11, 12]. Since the LZ4 is based on LZ77 compression algorithm, dictionary is used for compression. By using the dictionary, repeating sequence is replaced to token which consists of offset and match length. Based on these properties, LZ4 algorithm is used not only in Hadoop framework, but also in variable applications. The LZ4 algorithm is used in Solid-State Driver(SSD) to improve the storage performance and lifetime [13]. And LZ4 is also used in forex trading system to improve the transfer logging speed [12].

Although LZ4 compression is already adopted variable applications with high compression throughput, there is rooms for reforming the LZ4 algorithm to be applied to more applications. Kwon et al. reduced the memory size and manufacturing costs by optimizing the LZ4 compression algorithm [14]. They modified the scanning window to move 4 bytes, and induced changes in offset. As a result, they were able to reach the memory performance goal by not saving the least 2 bits of the offset value. Kim et al. optimized the LZ4 algorithm to fit the mobile devices for memory performance and power consumption [15]. Instead of adding headers to LZ4 data format, they improved compression ratio by reducing offset value size. Also, they optimized the hash computing and reduced hardware resource.

Implementing the compression algorithm on hardware accelerator is a suitable solution for improving the compression throughput. And several compression algorithms are realized on hardware accelerator and adopted on multiple applications. Jing et al. proposed the new lossless compression algorithm named bit-mapping and implemented the bit-mapping algorithm on hardware accelerator [16]. Qiao et al. implemented the BWT algorithm on Field Programmable Gate Array (FPGA) for Bzip2 compression [17]. They achieved the 2 times over fast compression speed compared to software implementation. Du et al. proposed deflate compression accelerator to improve the disk access efficiency [4]. They designed the deflate compression accelerator and analysis the performance by using several test tools such as zpipe, TestDFSIO, and Terasort. By using the compression accelerator, they improved the compression throughput than software-only solution.

LZ4 compression algorithm is also being studied in hardware implementation to adopt on the hardware accelerator. Benes Tomas implemented the LZ4 algorithm on an FPGA and analysis the compression ratio, throughput, and utilization [18]. Bartik et al. realized the LZ4 on an FPGA. They verified the possibilities of realization on hardware by comparing area, frequency performance of LZRW (Lempel-Ziv Ross Williams) [19]. Jang et al. improved the transfer logging speed of the forex trading system by implementation of the LZ4 algorithm on hardware compression accelerator [12]. However, the study of the LZ4 algorithm to address the power and cost issues of Hadoop is still needed. Based on our previous works [20, 21], we aim to address the TCO and cost issues of Hadoop by adopting the LZ4 hardware accelerator on Atom-based server.

4. Lempel-Ziv 4

4.1 Compression algorithms used in Hadoop

Servers of the Hadoop framework compress the data to reduce the overhead of the data I/O, which saves storage of server, and improves transmission bandwidth. Hadoop includes Bzip2, Zlib, LZO, Zstandard, Snappy, and LZ4 compression algorithms, and these algorithms have different compression performance. Before we design the compression hardware accelerator, we evaluated the compression performance of these algorithms.

Compression throughput and compression ratio was analyzed to find the right algorithm for offloading. For performance analysis of compression algorithms, Intel Core i5-4590 CPU @ 3.30GHz x 2, 3.8 GiB RAM, Ubuntu 16.04.6 LTS OS was used. And fifteen suitable text files were used for corpus, which derived from the calgary, canterbury, snappy-master1, and silesia. There were subtle differences in the performance of compression algorithms depending on the corpus. **Table 1** shows the average of fifteen compression results to measure the overall compression ratio (original/compressed).

Table 1. Compression results of Hadoop compression algorithms

Compression algorithm	Original size (byte)	Compressed size (byte)	Compression ratio	Compression throughput (MB/s)	
				Compression	Decompression
LZO	1,061,790	486,971	2.180	27.867	383.400
Zstandard	1,061,790	332,882	3.190	24.267	421.200
Bzip2	1,061,790	281,915	3.766	12.003	35.733
Snappy	1,061,790	636,059	1.669	300.333	1,058.400
Zlib	1,061,790	386,220	2.749	18.609	272.467
LZ4	1,061,790	642,661	1.652	402.800	3,100.200

Bzip2 showed the highest compression ratio with the 3.766. Burrows-Wheeler transform and Huffman coding may contribute to the positive result of compression ratio. Zstandard with the value of 3.190 and Zlib with the value of 2.749 showed the highest compression ratio after Bzip2. LZ4 has the lowest value among the compression algorithm with 1.652 compression ratio. However, the result of compression/decompression throughput in **Table 1** showed the opposite tendency of compression ratio results. The Bzip2 and Zstandard compression algorithm, which had a highest compression ratio, recorded the lowest compression throughput. The LZ4 algorithm showed the highest throughput among the Hadoop compression algorithms with the value of 402.8 MB/s compression throughput and 3,100.2 MB/s decompression throughput. As shown in **Table 1**, there is a trade-off between compression ratio and throughput. Therefore, the compression algorithm adopted in a certain environment can be changed depending on the application needs. According to the performance analysis, LZ4 compression algorithm is suitable for hardware accelerators with the low-power processor. The high compression throughput of LZ4 can reduces the workload of the server by allowing the server to compress/decompress data faster. Also, hardware accelerator based on LZ4 has the advantage of hardware resource, because LZ4 compression algorithm is suitable to implement on hardware compared to other compression algorithms. Therefore, we adopted the LZ4 algorithm for the compression hardware accelerator in consideration of the high compression throughput and the advantage of hardware resource.

4.2 LZ4 compression Algorithm

LZ4 compression algorithm is suitable to implement on a hardware accelerator compared to other compression algorithms, as LZ4 is based on the dictionary-based compression algorithm. Algorithm 1. shows the encoding flow of the LZ4 frame. When LZ4 compression starts, the original sequence is moved to the window in order. LZ4 window includes the part of the original sequence which is compared with the dictionary. LZ4 window is classified as a current window, a lookahead window, and a scanning window. The current window contains the sequence for which compression will be performed, and the lookahead window contains the sequence to be shifted with the current window when compression of the current window is completed. The scanning window moves 1 byte from the current window to confirm the match between the sequence and dictionary. The size of scanning window equals in size to current window and lookahead window. In LZ4 compression algorithm, the first 4 bytes of scanning window are used to calculate the hash value to proceed with the match. As LZ4 frame has at least 3 bytes, more than 4 bytes of original sequence have to be compressed. Because the original sequence less than 4 bytes cannot be reduced by LZ4 compression. Also, computing the hash value of all the characters of the scanning window may occur calculation overhead. Therefore, 4 bytes of the scanning window are used to obtain the bucket value through the hash function, and the bucket is used as the address of the dictionary to perform the match procedure. When the hash value is not matched, the substring is registered at the address of the dictionary and the scanning window shifts 1 byte to repeat the above process at the next position of the current window. On the contrary, when the match is occurred, the matched length is calculated by backward match as the 4 bytes of the scanning window and dictionary are containing the same substring. After completing the backward match, the scanning window is shifted 1 byte to proceed the above process in the new character until the matched length of all characters in the current window is obtained. After then, based on the dictionary data which has the longest match length among the all characters of the current window, LZ4 data block is created by combining offset, compression, and uncompressed literal of the sequence. Then, the sequence in the lookahead window is shifted to the current window to create the next block, and the above compression process is repeated. LZ4 compression algorithm has exceptional rules. The match will not proceed, when the original sequence remains less than 12 characters. Also, last 5 characters in the original sequence will be left as uncompressed literal.

Algorithm 1. The encoding algorithm of LZ4 frame

Input:
p_i : pointer of input data
p_o : pointer of output data
Output:
b_o : buffer of output data
1. initialize p_i, p_o, b_o
2. **while** $p_i < size_i - 12$ **do**
3. $d_{in} \leftarrow$ read_sequence(p_i)
4. $h_{in} \leftarrow$ hash_function(d_{in})
5. $h_{data} \leftarrow$ hash_table[h_{in}]
6. hash_table[h_{in}] $\leftarrow$ d_{in}
7. **if** $h_{data} = d_{in}$ **then**
8. calculate match_length
9. encode lz4_frame
10. **for** i $\leftarrow$ 0 **to** lz4_frame_length – 1 **do**

```
11.                         b_o[p_o + i] ← lz4_frame[i]
12                      end
13.                     p_i ← p_i + match_length
14.                     p_o ← p_o + sequence_length
15.             else
16.                     p_i ← p_i +1
17.             end
18.     end
19.     return b_o
```

4.3 LZ4 Framing Format

After compress the original sequences, compressed data is reformed as LZ4 framing format which reformed according to the presence of flags or compressed data size. As shown in **Fig. 2**, the LZ4 framing format consists of a *magic number*, a *frame descriptor*, a *data block*, an *endmark*, and a *content checksum*. The *magic number* is 4 bytes little endian format and the value is fixed to 0x184D2204. The *frame descriptor* has minimum 3 to maximum 15 bytes data length depending on the optional parameter. Also, the *frame descriptor* contains a *flags*, a *content size*, and a *dictionary ID* that set the LZ4 frame format. The *data blocks* consists of a *compressed data*, a *block size*, and a *block checksum*. The functions of each part will be explained later on. When the *data block* has the 0 value, the part of the *data blocks* is terminated and the *endmark* representing the number of the *data block* is followed. The *content checksum* is the part that verifying the decoded value is correct. The *content checksum* is presented when the *content checksum* flag is asserted. The receiver can verify the correctness of LZ4 framing format by using the *content checksum*, and therefore, using the *content checksum* is encouraged.

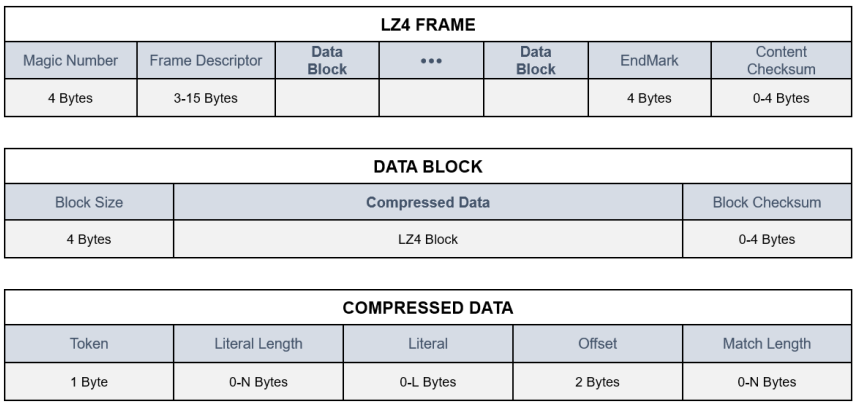


Fig. 2. LZ4 frame format

The structure of the *data blocks* is shown in **Fig. 2**. The *data block* consists of a *block size*, a *compressed data*, and a *block checksum*. The *block size* has 4 bytes little endian format, and the *compressed data* contains the uncompressed original sequence, when the highest bit is set to 1. On the contrary, when the highest bit is 0, the *compressed data* contains compressed sequence. In addition, the remaining bits in the *block size* have the size information of the following data block excepting the *block checksum*. The *Compressed data* contains the compressed data by using LZ4 algorithms, and properties of the *compressed data* will be explained later on. The *Block checksum* is only presented when the *block checksum* flag that included in the *frame descriptor* is set. As in the case of the *content checksum*, the *block*

checksum outputs the checksum value of the raw data block. The checksum is represented as little endian format.

The *compressed data* containing compression results of LZ4 algorithm consists of a *token*, a *literal length*, a *literal*, an *offset*, and a *match length*. The *Token* has a single byte size, and each upper and lower of 4 bits is classified as literal part and match part according to the function. The *Literal* part contains the number of uncompressed character in the window. When the length of data is more than 15, the more bits are needed to contain the number of uncompressed characters since the *literal* part only has 4 bit. In that case, the *literal* part can express 255 more length by using the 1 byte *literal length* part, and there is no limitation to add the *literal length* parts. The lower 4 bits of *token* are *match* part which represents the number of compressed characters. As in the case of *literal* part, *match* part express up to 15 length and the *match length* can be used to express more than 15 length. The *literal* is the part that containing the uncompressed data at each data and the *literal* has little endian format. The *offset* part represents the distance between the sequence of compressed data and registered data in the dictionary. The range of the offset can be restricted to one data block or entire data blocks according to the *block independence flag* of the *frame descriptor*.

4.4 Dictionaries in LZ4 Algorithm

LZ4 has the highest compression throughput among the Lempel-Ziv based algorithms. **Fig. 3** illustrates the encoding process of LZ4 algorithm. In the LZ4 compression, the input stream is scanned with the window which has 4 byte length and checked whether the substring was repeated in the input stream before. The LZ4 hash table is used to check the input steram. LZ4 hash table contains the substrings and indexes. The substring is compared with input stream and index has the position information of the input stream. When the substring of the hash table is equal to the current window, it means that the current substring is repeated, as shown in Fig. 3. In the LZ4, the match is the procedure that finding the repeated string from the input stream and calculate the total length of the repeated string. The match procedure is iterated for all the substrings in the LZ4 window and the longest match is calculated. The token is generated based on the information of longest match. The token (10,7) describes that there are 10 bytes of uncompressed literals and 7 compressed data bytes. The offset value 9 represents that the literals, matched with previously compressed literals, have been appeared before the offset value. When the entry doesn't exist in the hash table, a new entry is added to the hash table. The window scans and repeats this sequence to the end of the stream. As LZ4 hash table contains the small size bucket without the additional addressing functions, large size of memory is unnecessary for LZ4 algorithm.

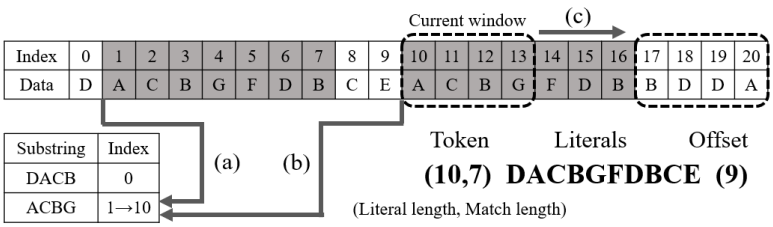


Fig. 3. The flow of generating compressed data

5. ARCHITECTURE OF THE ACC-LZ4

5.1 Micro-Architecture of ACC-LZ4

The **Fig. 4** illustrates the micro-architecture of the ACC-LZ4. The ACC-LZ4 is the hardware accelerator designed for LZ4 compression algorithm. Our hardware accelerator is composed of as follows: an *allocator*, 16 *dictionaries*, a *compare match*, a *compressed data write*, a *manager*, a *position*, and a *FSM* module. The *allocator*, including sliding windows, performs the data allocation to other modules. Then, it allocates sequences to each dictionary. The *dictionaries* compare allocated sequence with existing sequence to find out the repeated data strings. Every *dictionary* is connected to work signal, which is matched with 1 bit for each dictionary to control the dictionary separately. We can achieve the parallelism by deploying the *dictionaries* as parallel for compression throughput performance. The *compare match* module finds the longest match length from each *dictionary* through 4 compare stages. Then, it deploys the best compression result. This is the most significant principle in the LZ4 compression algorithm. The *compressed data write* module builds the LZ4 data frames by using the compression result, which is matched in *compare match* module. After then, it is stored in the output buffer. Therefore, the *compressed data write* module covers the uncompressed literals. The uncompressed literals are temporarily saved into the internal buffer until the occurrence of the match and LZ4 header. For the fine compression ratio, enough buffer size would be better. However, the buffer size is limited to output the uncompressed literals for the stall-free architecture.

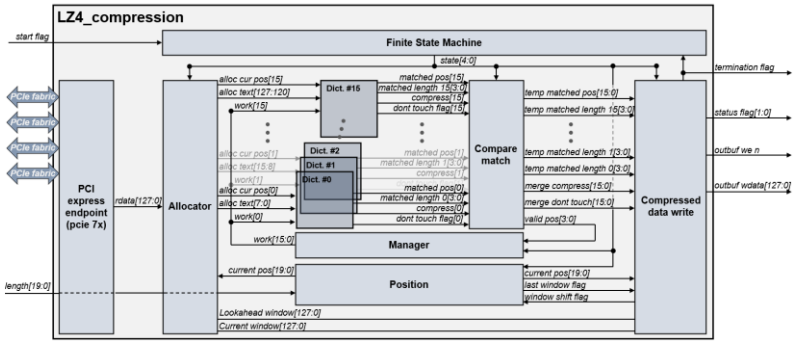


Fig. 4. Micro-Architecture of ACC-LZ4

The **Fig. 5** illustrates the fully pipelined architecture of the proposed hardware accelerator. The ACC-LZ4 receives 16 bytes of data from its input source every cycle and directs them into our stall-free latency pipeline. Thanks to the no-stall architecture, our hardware compression accelerator has (16 bytes×# of cores)/(20 cycles×period) compression throughput. The proposed architecture is composed of four major functional components: fetch, candidate match, match selection, and write. The operation of each stage is as follows:

- Fetch: The input sequence is slid into the current window from the lookahead window. The current window indicates the sequence processed in current iteration and the lookahead window indicates sequence text processed in next iteration. The parallel sequence is prepared from the fetch stage while other stages are conducted.
- Candidate match: The parallel sequence is compared with each dictionary for candidate match, where sixteen match lengths data are calculated.

- Match selection: The longest match length is found among the sixteen match results to obtain the best compression ratio. LZ4 frame is encoded by the token, literals, match length, and offset of the dictionary data which has the best match result.
- Write: The compressed data is fed to the output buffer through the write logic. The write stage has an extra FSM.

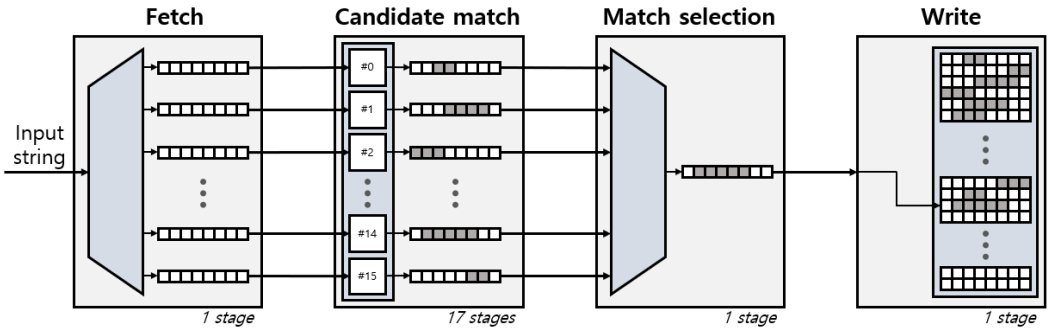


Fig. 5. Pipelined stage of ACC-LZ4

5.2 Parallel Dictionary in the ACC-LZ4

The ACC-LZ4 is based on the parallel dictionary architecture for efficient compression in hardware implementation. In the LZ4 algorithm, the dictionary finds the first match between the inside dictionary data and current window data. By using the window for match procedure, the ACC-LZ4 compresses the length of data up to 31 bytes. The first match process is a major cause of overhead. Therefore, we designed the dictionaries to reduce the compression time and to parallelize the LZ4 encoding. The proposed dictionary has a short bucket bit to reduce the complexity of the hash function. We used ASCII as the hash address, where the first character of the current window data is used. Thanks to the parallel dictionary, we can achieve the high compression throughput by exploiting sixteen dictionaries in parallel. When the compression hardware accelerator is designed with a single dictionary, the compression engine repeats the match procedure. On the contrary, our compression accelerator simultaneously compares current window data with 16 dictionaries. Thus, ACC-LZ4 achieves the higher throughput than a single dictionary-based architecture.

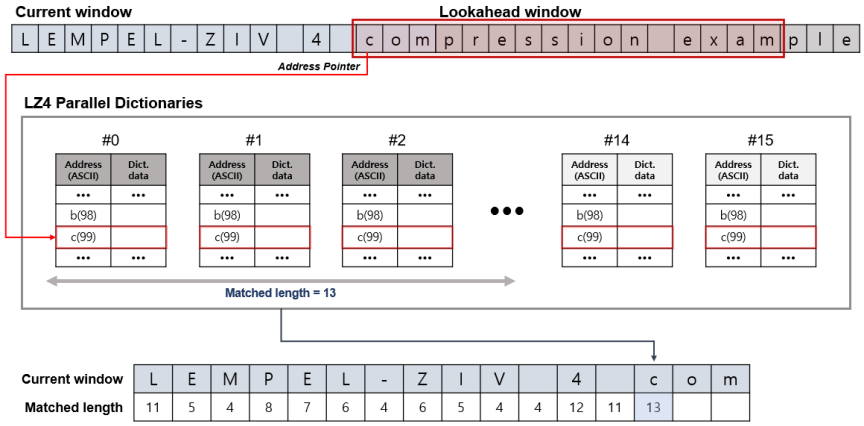


Fig. 6. Parallel dictionaries of ACC-LZ4

6. Performance Analysis

6.1 Environment

To evaluate the compression performance, we compared the compression throughput between the software LZ4 and ACC-LZ4 hardware accelerator. We used LZ4 open source code to evaluate the performance of software-based LZ4. And we set the evaluation environment to Intel Core i5-4590 CPU @ 3.30GHz x 2, 3.8 GiB RAM, and Ubuntu 16.04.6 LTS OS to evaluate the software-based LZ4. The ACC-LZ4 has 100 Mhz system clock, and comprises of eight LZ4 cores. To realize the LZ4 hardware accelerator, the ACC-LZ4 was implemented on Artix7, Kintex7, Virtex7, and Zynq7000. Also, logic analyzer was used to evaluate the time between the compression start and termination flag.

6.2 Experimental Dataset

The experimental dataset is selected for comparing the performance of ACC-LZ4 and software-based LZ4. As the LZ4 is a byte-oriented algorithm, text files are used as corpus. We selected 15 text files as corpus from calgary, canterbury, silesia, and snappy-master1. The compression ratio was different as text file, but the results are generally similar.

6.3 Experimental Results

The experiment was conducted to evaluate the compression throughput. In order to compare the throughput performance between ACC-LZ4 and software-based LZ4, we analyzed the compression time of each corpus sequence. For the experiment, we intentionally generated the compression start and compression termination I/O signals on the ACC-LZ4. With the logic analyzer, we measured the compression time by using the compression start and compression termination signals. **Table 2** shows the compression throughput of software-based LZ4 and ACC-LZ4. The results show that the different aspects of compression throughput between software-based LZ4 and ACC-LZ4. When text file is compressed by software-based LZ4, every compression throughput of corpus is fairly different because iteration can be terminated irregularly as the software-based LZ4 finds the match with multiple iterations. On the contrary, the ACC-LZ4 has the fairly constant compression throughput compared to software LZ4, because 16 parallel dictionaries are used at the match operation. Consequently, software-based LZ4 has 410MB/s of compression throughput and ACC-LZ4 shows 639MB/s of compression throughput in average. The ACC-LZ4 has 1.558 times faster compression throughput compared to software-based LZ4 because the ACC-LZ4 is designed to solve the iteration overhead issue by using the parallel dictionaries and ASCII hash function.

Table 2. Compression results of software-based LZ4 and ACC-LZ4

Input text	Original size (byte)	Compressed size (byte)	Compression ratio	Compression throughput (MB/s)	
				Software based LZ4	ACC-LZ4
alice29.txt	152,089	88,699	1.715	363	640.517
news	377,109	222,770	1.693	426	640.183
asyoulik.txt	125,179	79,653	1.572	372	639.958
bib	111,261	56,688	1.963	406	640.037
book1	768,771	522,806	1.470	344	640.227
book2	610,856	333,498	1.832	366	640.088
lcet10.txt	426,754	233,213	1.830	366	639.524
paper1	53,161	28,933	1.837	399	639.229

paper2	82,199	47,824	1.719	371	640.086
paper3	46,526	28,294	1.644	363	639.107
plrabn12.txt	481,861	325,589	1.480	347	640.018
random.txt	100,000	100,394	0.996	403	639.427
paper4	13,286	8,471	1.568	574	637.009
paper5	11,954	7,456	1.603	610	636.296
paper6	38,105	20,609	1.849	435	639.611
Average	226,607	140,326	1.651	410	639.421

In order to verify the resource of the ACC-LZ4, we implemented the ACC-LZ4 on the variable FPGAs. We implemented the ACC-LZ4 on the Virtex7 (XC7VX485TFFG176), Kintex7 (XC7K325TFFG900), Artix7 (XC7A200TSBG484), and Zynq7000 (XC7Z010CLG400). **Table 3** is the utilization results of the ACC-LZ4. The used resources of the slices, LUTs, and Flip-Flops of each FPGAs are presented on **Table 3**. Based on the results of the implementation, we verified that the ACC-LZ4 can be realized on the low-cost FPGA such as Zynq7000 and it will lessen the workload of server and reduce the cost of the Hadoop storage. Based on these results, data I/O overhead of Hadoop can be reduced by using the low-performance processor with ACC-LZ4. Also, we verify that the ACC-LZ4 has higher compression throughput compared to high-performance processor. This strategy addresses the power and cost issue of Hadoop by using the low-performance processor with ACC-LZ4.

Table 3. Utilization results of the ACC-LZ4

FPGA	Slice	LUT	Memory	Flip-Flop
Virtex7	2,506 (3.30%)	7,099 (2.33%)	896 (0.68%)	2,230 (0.36%)
Kintex7	2,724 (5.34%)	7,155 (3.49%)	896 (1.40%)	2,230 (0.54%)
Artix7	2,488 (7.43%)	7,129 (5.32%)	896 (1.93%)	2,230 (0.83%)
Zynq7000	2,301 (52.29%)	6,883 (39.10%)	908 (15.13%)	2,110 (5.99%)

7. Conclusion

In this paper, we addressed the power and cost issues of Hadoop. We examined these issues derived from the high-performance processor and proposed the low-performance processor with the ACC-LZ4 compression hardware accelerator. When realize the ACC-LZ4 hardware accelerator to adopt on low performance processor, there are still issues of hash function and compression iteration. The hash function of LZ4 increases hardware resources of the accelerator, and compression iteration of LZ4 algorithm causes degradation of compression throughput. Thus, we addressed the hash function issue by reforming the ASCII based hash function and the iteration issue by adopting parallel dictionaries. The ACC-LZ4 has the 639MB/s compression throughput and has 1.558 times faster compression performance compared to software-based LZ4. Based on the ACC-LZ4, high-performance processor can be replaced with low-performance processor by offloading the load of compression operation to the ACC-LZ4. Consequently, adopting the ACC-LZ4 with low-performance processor address the TCO and cost issues of Hadoop framework. In the future, we plan to optimize the hardware resources of ACC-LZ4 and adopt more LZ4 cores on ACC-LZ4 to increase the compression throughput.

References

[1] D. Park, J. Wang, and Y. S. Kee, "In-Storage Computing for Hadoop MapReduce Framework Challenges and Possibilities," *IEEE Transaction on Computers*, p. 1, July 2016. Article (CrossRef Link)

[2] N. M. F. Qureshi and D. R. Shin, "RDP: A storage-tier-aware Robust Data Placement strategy for Hadoop in a Cloud-based Heterogeneous Environment," *KSII Transaction on Internet and Information Systems*, vol. 10, no. 9, Sep. 2016. Article (CrossRef Link)

[3] H. Xu, W. Liu, G. Shu, and J. Li, "LDBAS: Location-aware Data Block Allocation Strategy for HDFS-based Applications in the Cloud," *KSII Transaction on Internet and Information Systems*, vol. 12, no. 1, Jan. 2018. Article (CrossRef Link)

[4] H. Du, K. Zhang, S. Sha, C. Ye, and Q. Luo, "The Library for Hadoop deflate compression based on FPGA accelerator with Load Balance," in *Proc. of 2019 20th International Conference on Parallel and Distributed Computing, Applications and Technologies(PDCAT)*, pp. 265-270, 2019. Article (CrossRef Link)

[5] Y. Li, Y. Sun, G. Dai, Y. Wang, K. Ni, Y. Wang, G. Li, and H. Yang, "A Self-aware Data Compression System on FPGA in Hadoop," in *Proc. of International Conference on Field Programmable Technology (FPT)*, pp. 196-199, Dec. 2015. Article (CrossRef Link)

[6] L. H. Xiang, L. Miao, D. F. Zhang, and F. P. Chen, "Benefit of Compression in Hadoop: A Case Study of Improving IO Performance on Hadoop," in *Proc. of the 6th International Asia Conference on Industrial Engineering and Management Innovation*, pp.879-890, 2016. Article (CrossRef Link)

[7] S. Ibrahim, D. Moise, H. E. Chihoub, A. Carpen-Amarie, L. Bouge, and G. Antoniu, "Towards Efficient Power Management in MapReduce: Investigation of CPU-Frequencies Scaling on Power Efficiency in Hadoop," in *Proc. of International Workshop on Adaptive Resource Management and Scheduling for Cloud Computing*, pp.147-164, 2014. Article (CrossRef Link)

[8] N. Zhu, X. Liu, and Y. Hua, "Towards a cost-efficient MapReduce: Mitigating power peaks for Hadoop clusters," *Tsinghua Science and Technology*, vol. 19, no. 1, pp. 24-32, 2014. Article (CrossRef Link)

[9] Intel Atom Processor. [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/atom.html

[10] Intel Xeon Processor. [Online]. Available: https://www.intel.com/content/www/us/en/products/processors/xeon.html

[11] S. M. Lee, J. H. Oh, J. H. Jang, S. M. Lee, K. Kim, and S. E. Lee, "Live demonstration: An FPGA based hardware compression accelerator for Hadoop system," in *Proc. of IEEE Asia Pacific Conference on Circuits and Systems (APCCAS)*, pp. 744-745, 2016. Article (CrossRef Link)

[12] J. H. Jang, S. M. Lee, S. D. Kim, S. G. Oh, E. Ko, S. M. Lee, J. W. Shin, and S. E. Lee, "Accelerating Forex Trading System Through Transaction Log Compression," in *Proc. of International SoC Design Conference (ISODC)*, pp. 74-75, 2014. Article (CrossRef Link)

[13] W. Liu, F. Mei, C. Wang, M. O'Neill, and E. E. Swartzlander, "Data Compression Device Based on Modified LZ4 Algorithm," *IEEE Transactions on Consumer Electronics*, vol. 64, no. 1, pp. 110-117, 2018. Article (CrossRef Link)

[14] S. J. Kwon, S. H. Kim, H. J. Kim, and K. S. Kim, "LZ4m: A fast compression algorithm for in-memory data," in *Proc. of IEEE International Conference on Consumer Electronics (ICCE)*, pp. 420-423, Jan. 2017. Article (CrossRef Link)

[15] J. Kim and J. Cho, "Hardware-accelerated Fast Lossless Compression Based on LZ4 Algorithm," in *Proc. of International Conference on Digital Signal Processing*, pp. 65-68, Feb. 2019. Article (CrossRef Link)
[16] Y. Jing, L. Rong, G. Rui, and X. Ning-Yi "An Efficient Lossless Compression Method for Internet Search Data in Hardware Accelerators," *WRI World Congress on Computer Science and Information Engineering*, pp. 453-457, Apr. 2009. Article (CrossRef Link)
[17] W. Qiao, Z. Fang, M. C. F. Chang, and J. Cong, "An FPGA-based BWT Accelerator for Bzip2 Data Compression," in *Proc. of IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 1-4, May 2019. Article (CrossRef Link)
[18] B. Tomas, "High throughput FPGA implementation of LZ4 algorithm," M.S. thesis. Czech Technical University, Prague, 2019.
[19] M. Bartik, S. Ubik, and P. Kubalík, "LZ4 compression algorithm on FPGA," in *Proc. of IEEE International Conference on Electronics, Circuits, and Systems*, pp. 179-182, 2015. Article (CrossRef Link)
[20] S. M. Lee, J. H. Hang, J. H. Oh, J. K. Kim, and S. E. Lee, "Design of Hardware Accelerator for Lempel-Ziv 4(LZ4) Compression," *IEICE Electronics Express*, vol. 14, no. 11, 2017. Article (CrossRef Link)
[21] S. D. Kim, S. M. Lee, J. H. Jang, J. G. Son, Y. H. Kim, and S. E. Lee, "Compression Accelerator for Hadoop Appliance," in *Proc. of International Conference on Internet of Vehicles*, pp. 416-423, Sep. 2014. Article (CrossRef Link)

Do Young Choi received the B.S. degree in Electronic Engineering from the Myongji University. He is currently a M.S. student in Electronic Engineering in Seoul National University of Science and Technology. His current research interests include digital system design, computer architecture, and hardware accelerator for compression.

Jung Hwan Oh received the B.S. and M.S. degree in the Department of Electronic Engineering at the Seoul National University of Science and Technology, Seoul, Korea, in 2017 and 2019, respectively. He is currently an engineer at division of S.LSI in Samsung Electronics. His research interests include computer architecture, System-on-Chip design and hardware multi-core scheduler design.

Ji Kwang Kim received the B.S. and M.S. degree in the Department of Electronic Engineering at the Seoul National University of Science and Technology, Seoul, Korea, in 2017 and 2019, respectively. He is currently an engineer at division of memory in Samsung Electronics. His research interests include SoC design and memory controller architecture.

Seung Eun Lee received the Ph.D. degree in electrical and computer engineering from the University of California, Irvine (UC Irvine) in 2008 and the B.S. and M.S. degrees in electrical engineering from the Korea Advanced Institute of Science and Technology (KAIST), Daejeon in 1998 and 2000, respectively. After graduating, he had been with Intel Labs., Hillsboro, OR, where he worked as Platform Architect. In 2010, he joined the faculty of the Seoul National University of Science and Technology, Seoul. His current research interests include computer architecture, multi-processor system-on-chip, low-power and resilient VLSI, and hardware acceleration for emerging applications.