A High-Throughput Hardware Accelerator for Lossless Compression of a DDR4 Command Trace

Jiwoong Choi, Boyeal Kim, Hyun Kim, *Member, IEEE*, and Hyuk-Jae Lee, *Member, IEEE*

***Abstract*— In a memory system, understanding how the host is stressing the memory is important to improve memory performance. Accordingly, the need for the analysis of memory command trace, which the memory controller sends to the dynamic random access memory, has increased. However, the size of this trace is very large; consequently, a high-throughput hardware (HW) accelerator that can efficiently compress these data in real time is required. This paper proposes a high-throughput HW accelerator for lossless compression of the command trace. The proposed HW is designed in a pipeline structure to process Huffman tree generation, encoding, and stream merge. To avoid the HW cost increase owing to high-throughput processing, a Huffman tree is efficiently implemented by utilizing static random access memory-based queues and bitmaps. In addition, variable length stream merge is performed at a very low cost by reducing the HW wire width using the mathematical properties of Huffman coding and processing the metadata and the Huffman codeword using FIFO separately. Furthermore, to improve the compression efficiency of the DDR4 memory command, the proposed design includes two preprocessing operations, the "don't care bits override" and the "bits arrange," which utilize the operating characteristics of DDR4 memory. The proposed compression architecture with such preprocessing operations achieves a high throughput of 8 GB/s with a compression ratio of 40.13% on average. Moreover, the total HW resource per throughput of the proposed architecture is superior to the previous implementations.**

***Index Terms*— Block Huffman, field-programmable gate array (FPGA), high-throughput hardware (HW) design, lossless compression, memory command trace analysis.**

I. Introduction

OWING to the increasing demand for big data and neural networks, the role of dynamic random access memory (DRAM) has become increasingly important and the memory bandwidth (BW) has rapidly increased. To understand how the host stresses the DRAM system and how the memory is vulnerable under certain situations, analyzing the memory access pattern [1] of how the system controls the DRAM is necessary. Memory performance depends on the memory access patterns [2], and memory trace analysis for these access patterns has been actively conducted by memory manufacturers to improve the DRAM performance [3]. However, as the memory command trace data have a throughput of 32 B/cycle based on the most commonly used DDR4 memory, the data size to be collected for analysis is very large [4]. As the cost of storing and analyzing such large amounts of data is considerably high, and the memory BW to be transferred in real time to the storage space is also very large; various compression studies [5]–[9] have been undertaken to reduce the trace data size.

For an efficient trace data compression, the following three conditions should be satisfied. First, lossless compression is essential as the loss of memory command trace data should not occur. Second, a hardware (HW) design with high speed and high throughput is required for compressing on-the-fly data with a high BW of 32 B/cycle in the high-speed system environment above 250 MHz. As data BW and operating frequency are gradually increasing in the memory system, high-throughput processing of 8 GB/s is generally required. This characteristic is even more important in recent years because memory structure such as high BW memory [10] is spotlighted for supporting the rapid increase in the amount of data. Therefore, trace analysis system essentially requires an HW-based compression accelerator instead of a software (SW)-based compression method. It is noteworthy that the SW-based compression is much slower than the HW-based compression, which degrades the overall system performance [11]. Third, HW cost and complexity should be low while maintaining high compression efficiency. In other words, the tradeoff between compression efficiency and HW resource must be carefully considered.

Lossless compression is classified into two categories: dictionary-based method and statistical method. The dictionary-based method, which is represented by the Lempel–Ziv series [12]–[15], compresses the data using the location information of the previous data. This method shows a relatively high compression ratio but is disadvantageous in that it requires large HW resources and complexity because the dictionary table must be created continuously for each input data by checking the dependence and location with previous data. On the other hand, the statistical method based on entropy, which is represented by the Huffman coding [16],

Manuscript received April 13, 2018; revised July 19, 2018; accepted August 27, 2018. Date of publication September 26, 2018; date of current version December 28, 2018. This work was supported in part by the Ministry of Trade, Industry, and Energy under Grant 10080613, in part by the Korea Semiconductor Research Consortium Support Program for the development of the future semiconductor device, and in part by SK Hynix Inc. *(Corresponding author: Hyun Kim.)*

J. Choi, B. Kim, and H.-J. Lee are with the Inter-University Semiconductor Research Center, Department of Electrical and Computer Engineering, Seoul National University, Seoul 08826, South Korea (e-mail: jwchoi@capp.snu.ac.kr; bykim@capp.snu.ac.kr; hyuk_jae_lee@capp.snu.ac.kr).

H. Kim is with the Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul 01811, South Korea (e-mail: hyunkim@seoultech.ac.kr).

Digital Object Identifier 10.1109/TVLSI.2018.2869663

1063-8210 © 2018 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

constructs a tree with the frequency of symbol occurrence and compresses data by allocating the codeword according to the tree. This method can be implemented with relatively low complexity, but it shows a relatively low compression ratio. To take advantages of each method and to compensate for the disadvantages of each method, lossless compression schemes combining various compression algorithms have been widely studied [17]–[20], and the HW implementation of these lossless compression techniques has also been actively studied [21]–[24].

Nunez and Jones [21] propose an HW architecture of the X-MatchPro algorithm that compresses data at a throughput of 1.6 Gbit/s with high HW scalability. Lin *et al.* [22], [23] propose an HW structure that combines parallel dictionary Lempel–Ziv–Welch (PDLZW) and dynamic Huffman coding [25]. It offers the advantage of lower HW complexity and better compression efficiency than the dynamic Huffman. Fowers *et al.* [24] propose the HW architecture of the DEFLATE algorithm [17] that shows relatively high-throughput processing of 5.6 GB/s with high HW scalability. All these previous studies in [21]–[24] use Huffman coding, but they are not an optimal Huffman coding. Nunez and Jones [21] and Fowers *et al.* [24] used a predetermined tree structure; they cannot compress data while changing the tree structure according to the data. In case of Lin *et al.* [22], [23], they cannot process compression and tree formation at the same time because they build an approximated tree structure on an offline process. Thus, the compression ratio is inevitably sacrificed. Furthermore, the operating frequency of these previous schemes is relatively low and except [24], they can only process the input data with a small BW, rendering it difficult to process high-throughput data of 8 GB/s. This means that these previous schemes are incomplete in terms of compression efficiency and HW complexity/cost.

Herein, to compensate for this drawback, the tradeoff between compression efficiency and HW cost is carefully considered through the efficient pipeline processing of block-based compression and HW-optimized design techniques. More specifically, this paper proposes an HW compression accelerator architecture capable of high-throughput processing based on the block Huffman coding [26]. The block Huffman coding is a relatively simple statistical compression method that processes Huffman coding by dividing the input data into blocks. Although the compression efficiency of the block Huffman is slightly lower than that of the original Huffman coding, the two-pass operation in the original Huffman coding can be performed by one-pass operation through the block-based pipeline structure. Moreover, because the frequency of the symbol occurrence is limited by the block size, the worst case of the Huffman codeword length can be significantly reduced [27]. Therefore, it is possible to drastically reduce the HW resources and process the high-throughput on-the-fly data compression. In the HW implementation of Huffman coding, the Huffman tree formation and merge processing of variable length coding are very difficult. To address these problems, the proposed HW architecture efficiently generates a new tree every time according to the input data using static random access memory (SRAM)-based queues and bitmaps and processes the variable length stream at a low cost using FIFO and HW wire width optimization. All of these processes are pipelined to achieve a high throughput of 8 GB/s. In addition, as the Huffman tree is generated for every block, it has a higher compression ratio than that of the predefined Huffman tree and approximated Huffman tree. Consequently, the proposed HW is excellent in terms of the tradeoff between HW cost and throughput. From these reasons, in this paper, block Huffman coding, which satisfies the second (i.e., high throughput) and third (i.e., low HW cost) conditions mentioned above, has been selected for memory trace compression.

Although block Huffman coding satisfies other conditions perfectly, it cannot satisfy the condition that it should achieve high compression efficiency, which is a part of the third condition mentioned earlier. Therefore, in this paper, to further enhance the compression ratio for the memory command trace data, two preprocessing methods, i.e., "don't care bits override" on the device deselect commands, and "bits arrange" on the command data, are proposed. These schemes utilize the operating characteristics of the DDR4 memory. It is noteworthy that the proposed preprocessing techniques are compatible with the block Huffman coding because the complexity of the block Huffman coding is very low. Finally, this paper proposes a high-throughput HW accelerator of lossless compression with the high compression ratio that meets all three conditions mentioned earlier. The proposed HW exhibits an excellent compression ratio of 40.13% on average and its total HW resource per throughput is superior to the previous studies. It is noteworthy that the BW reduction of the memory command leads to the power reduction in the DRAM memory accesses.

The remainder of this paper is organized as follows. Section II describes the various Huffman coding schemes. Section III presents the proposed HW accelerator architecture, implementation issues, and preprocessing methods. Section IV compares the proposed system with other studies and demonstrates the superiority of the proposed compression HW. Section V concludes this paper.

II. Background

Huffman coding [16], a tree-based lossless data compression technique is the most representative entropy coding method. It is widely used in various fields requiring compressions such as systems-on-a-chip design [28] and image processing [29]. Huffman coding requires information on the frequency of each symbol in advance. The tree is constructed using the frequency of each symbol, and the most common symbol is in the upper node of the tree and expressed as the shortest codeword. However, Huffman coding cannot be applied to a continuous source stream because it cannot construct a Huffman tree for compressing an input stream unless the information for the entire stream is provided. To solve this problem, dynamic Huffman coding [25] has been proposed. This method can create a Huffman tree dynamically while the data are being inputted, even though the frequency information of the symbol is not provided in advance. In other words, this scheme can construct a tree by one pass. However, the complexity is too high because the tree must be relocated every time

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

the symbol is input and, consequently, it is not suitable for HW implementation.

To overcome the drawbacks of dynamic Huffman coding, block Huffman coding [26] has been proposed. This method can cope with the continuous input stream while maintaining the simplicity of Huffman coding. It processes the input stream in a block unit and applies Huffman coding to each block. The advantage of this scheme is that high-throughput compression is possible with low HW complexity because block processing and Huffman coding are performed independently. Huffman coding belongs to variable length coding and, consequently, the codeword length of each symbol is different depending on the frequency information of the symbol. Therefore, in the worst case of the conventional Huffman coding [16], an 8-bit symbol is represented by a codeword of 256 bits. However, block Huffman coding can significantly reduce the worst case Huffman codeword length because the frequency of symbols is limited to the block size [27]. The longest Huffman codeword length is expressed as follows:

$$\frac{1}{F_{K+3}} < p < \frac{1}{F_{K+2}}. \tag{1}$$

In the source data, the smallest probability value is defined as p, and F means the Fibonacci sequence value at the corresponding index. For example, if the block size of block Huffman coding is 16 kB, the probability value of the least-appearing symbol is 1/16K, and the K value satisfying (1) is 19. In other words, the worst case Huffman codeword length of block Huffman coding is 19 bits, which is considerably smaller than the worst case 256 bits of Huffman coding [16]. Owing to the reduced worst case of codeword length, the HW cost for variable length stream merge can be significantly reduced. In this regard, block Huffman coding is suitable for a low-cost HW accelerator of lossless compression with high throughput. An efficient implementation of block Huffman coding is presented in Section III-B.

However, as block Huffman coding divides data into block units, it creates a code tree using temporal locality rather than a whole. Therefore, block Huffman coding generally has a lower compression ratio than conventional Huffman coding. To compensate for these drawbacks, a more efficient system can be implemented by allocating additional resources to various preprocessing operations, taking advantage of that the HW resource of block Huffman coding is much smaller than that of conventional Huffman implementations. The preprocessing can reflect the characteristics of the input data and, thus, it is expected to achieve a very high compression efficiency. Section III-C discusses the preprocessing techniques that can be used to improve the compression ratio of memory command trace data, which are the target data of this paper.

III. High-Performance Lossless Compression

A. Overview of the Proposed Lossless Compression Hardware

Fig. 1 shows the overall structure of the proposed HW design. It consists of three parts: block Huffman, parallel stream merge, and preprocessing.

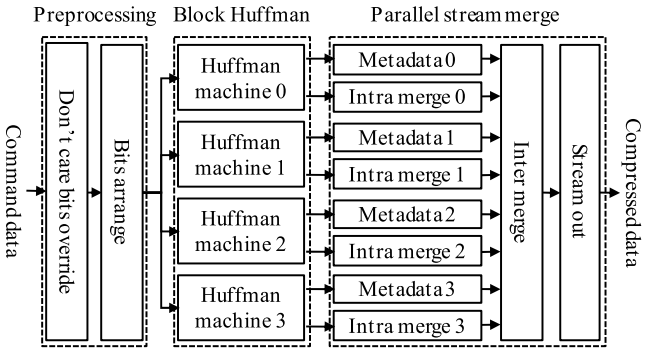


Fig. 1. Overall structure of the proposed hardware design.

The block Huffman module consists of a buffer, a Huffman front part, and a Huffman back part, as shown in Fig. 2. In the Huffman front part, the frequency count module calculates the symbol frequency of the input data and the sorting module arranges the symbols in the frequency order by merge sort. In the Huffman back part, a Huffman tree is formed using the symbols and frequencies sorted in the Huffman front part. In this process, the module uses SRAM-based queues and bitmaps appropriately to form the Huffman tree and the Huffman codeword effectively. A parallel stream merge module packs the output streams of four parallel compression machines into one stream. It consists of the metadata, intramerge, intermerge, and stream out module. In the metadata module, symbol and frequency information are packed into the single stream because they are needed to construct the tree at the decoding process. The intramerge module merges the eight variable length streams from the single compression machine into the single stream using shifters and OR operations. The intermerge module packs the metadata and intramerge streams into the single stream by shifters and OR operations. The stream out module slices the final compressed stream from the intermerge module to a certain size and sends it to the final output. This stream merge module is essential for variable length coding but requires tremendous HW cost for high-throughput data processing. To address this drawback, herein, the parallel stream merge module efficiently pipelines these merge processes and optimizes the HW wire width using the mathematical properties of Huffman coding, to enable high-throughput data processing with a very low HW cost. Consequently, the proposed HW accelerator performs lossless data compression with a high throughput of 8 GB/s and achieves excellent HW cost per throughput performance. Furthermore, it can preserve the compression ratio of optimal Huffman coding because it generates the new Huffman tree continuously according to the input data. This low-cost HW design of lossless compression, which maintains high compression efficiency and high throughput, is the first contribution of this paper and the implementation details of the proposed HW design are presented in Section III-B.

Even if the block Huffman coding module is implemented efficiently, compressing the DDR4 command trace data using Huffman coding results in very low compression efficiency because the DDR4 command trace data have a high entropy

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

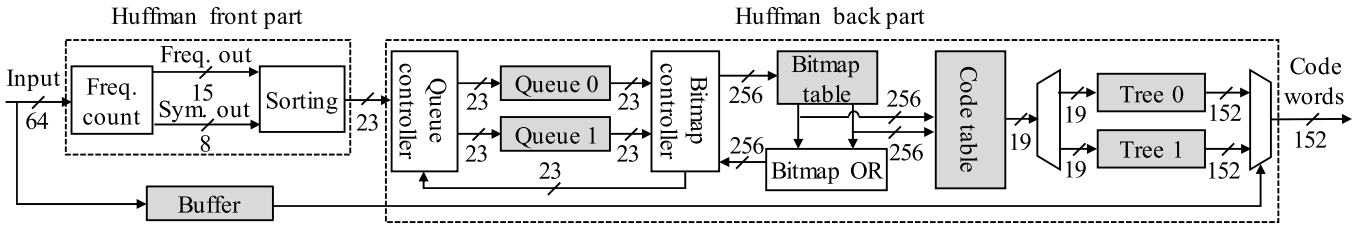


Fig. 2. Architecture of Huffman machine in block Huffman module. It only represents the data path, and the unit of wire is bit. Gray is implemented as SRAM. Buffer is a single port SRAM, and the others are dual port SRAMs.

owing to their complex structure and Huffman coding generally has a relatively lower compression ratio [30]–[32] than arithmetic coding [33] or dictionary-based coding. It is noteworthy that the pattern distribution of the DDR4 command trace data is irregular because these data consist of a combination of addresses and control signals and the information contained in each signal differs depending on the command type [34]. Herein, to achieve the high compression ratio for the DDR4 command trace data, two preprocessing schemes, the "don't care bits override" and the "bits arrange," which utilize the operating characteristics of the DDR4 memory, are proposed. The don't care bits override module can lower the entropy of data by overriding the don't care bits on the device deselect command to a value of 0 or 1. The bits arrange module can lower the entropy of data by dividing the memory command data composed of 32 bits into four groups of high correlation bits and obtain high compression ratio by compressing each partitioned data separately. It is noteworthy that the memory command data are divided into four groups because the block Huffman coding module compresses the memory command data in units of 8-bit symbols. These preprocessing modules are the second contribution of this paper and a detailed description of the preprocessing is covered in Section III-C.

B. Hardware Implementation

1) Block Huffman Coding: Fig. 2 shows the HW architecture of the Huffman machine in the block Huffman module. The preprocessed data are stored in the buffer for a certain block size, while it is sorted in the Huffman front-part module. If the input data are sorted by frequency, the time complexity of the tree composition [35] can be reduced from $O(n \log n)$ to $O(n)$ using two queues. To achieve this, the finite-state machine (FSM) with six states, as shown in Fig. 3, is used. As sorting is complete in the Huffman front part, the FSM state moves from the IDLE to the INIT and the sorted symbols and frequencies are stored in the first queue. After storing, FSM goes to the EXTRACT state and dequeues two data from each queue. Subsequently, in the COMPARE state, two data with low frequency are determined from the four data extracted from the queue. In the SELECT state, the symbol with the lowest frequency is selected as the left child, and the remaining is selected as the right child. If the frequencies are the same, the symbol that appears later in the ascending sort becomes the left child. Finally, in the INSERT state,

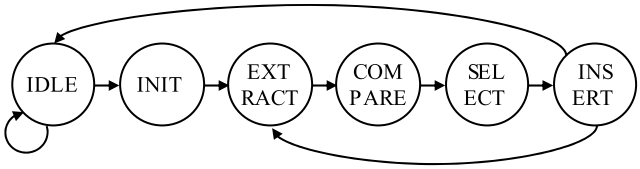


Fig. 3. FSM for block Huffman coding.

the parent data, which is the sum of the left child frequency and the right child frequency, is stored in the second queue. As the data stored in two queues becomes one, FSM returns to the IDLE state. Otherwise, the operation above is repeated. These processes are performed through the queue controller, queues, and bitmap controller, as shown in Fig. 2.

An SRAM-based queue is used to obtain the left child, right child, and parent data. The bitmap is used to efficiently allocate the Huffman codeword to the obtained data. Fig. 4 shows an example of the Huffman code table formation using a bitmap. After sorting in the Huffman front-part module, the FSM state moves from IDLE to INIT and the bitmap table is initialized during that state. Fig. 4(a) shows the results in the INIT state. In the EXTRACT state, the bitmap values stored in the SRAM are extracted using the left child and right child symbols that were obtained at the previous FSM loop. This process is controlled by the bitmap controller in Fig. 2. At the COMPARE state, the left child bitmap and the right child bitmap are used as addresses to form a code table. In the code table, the left child and right child are assigned 0 and 1, respectively. Furthermore, the code size of the corresponding symbol is increased. In this process, the existing Huffman code in the code table needs to be shifted and merged with the new value before being stored in the code table. Because these operations are difficult to be processed in a single cycle, the code table is designed to process these operations during four cycles of COMPARE, SELECT, INSERT, and EXTRACT considering the FSM state that determines left child and right child through a queue. In other words, up to four of Huffman codes can be formed during four cycles on one dual port SRAM through pipeline processing. Therefore, the code table of the proposed HW is implemented with 64 dual port SRAM (each dual port SRAM consists of 4×19 bits) so that the worst case of the Huffman code formation (i.e., 256 symbols) can be processed in parallel. Fig. 4 (b)–(d) shows the code allocation results for each next COMPARE state. In the SELECT state, the OR result of the left child bitmap and the right child bitmap

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

Queue 0		Queue 1		Bitmap table		Code table		
Sym.	Freq.	Sym.	Freq.	Sym.	Bitmap	Sym.	Code	Size
A	10			E	1 0 0 0 0	E		
C	3			D	0 1 0 0 0	D		
D	3			C	0 0 1 0 0	C		
E	3			B	0 0 0 1 0	B		
B	2			A	0 0 0 0 1	A		

(a)

Queue 0		Queue 1		Bitmap table		Code table		
Sym.	Freq.	Sym.	Freq.	Sym.	Bitmap	Sym.	Code	Size
				E	**1 0 0 1 0**	**E**	**1**	**1**
				D	0 1 0 0 0	D		
A	10			C	0 0 1 0 0	C		
C	3			B	0 0 0 1 0	**B**	**0**	**1**
D	3	**E**	**5**	A	0 0 0 0 1	A		

(b)

Queue 0		Queue 1		Bitmap table		Code table		
Sym.	Freq.	Sym.	Freq.	Sym.	Bitmap	Sym.	Code	Size
				E	1 0 0 1 0	E	1	1
				D	0 1 0 0 0	**D**	**0**	**1**
				C	**0 1 1 0 0**	**C**	**1**	**1**
		C	**6**	B	0 0 0 1 0	B	0	1
A	10	**E**	**5**	A	0 0 0 0 1	A		

(c)

Queue 0		Queue 1		Bitmap table		Code table		
Sym.	Freq.	Sym.	Freq.	Sym.	Bitmap	Sym.	Code	Size
				E	1 0 0 1 0	**E**	**1 0 1**	**3**
				D	0 1 0 0 0	**D**	**0 1 1**	**3**
				C	**1 1 1 1 1**	**C**	**1 1 1**	**3**
				B	0 0 0 1 0	**B**	**0 0 1**	**3**
		C	**21**	A	0 0 0 0 1	**A**	**0**	**1**

(d)

Fig. 4. Example of a Huffman code table formation. (a) Results in the INIT state. (b)–(d) Code allocation results for each next COMPARE state.

is updated in the bitmap table. Only the bitmap corresponding to the right child is updated because the left child symbol no longer appears in the queues. Thus, the SRAM-based bitmap can represent the tree structure effectively and create an HW-efficient Huffman code table. These processes are performed through the bitmap controller, bitmap OR, bitmap table, and code table, as shown in Fig. 2.

All of these processes are pipelined to achieve high throughput. The input data are stored using a triple buffer, and the generated Huffman code table is stored using a double buffer called a tree. In the encoding, eight symbols must be processed per cycle; therefore, one tree consists of four SRAMs. As the tree storage is completed, the Huffman codeword is extracted from the SRAM called tree using the input data value stored in the buffer. As a result, the proposed block Huffman coding is very well designed compared to the previous design [36] due to the proposed techniques (i.e., using the SRAM-based queues and bitmaps and processing each process in a pipelined manner).

2) Parallel Stream Merge: Fig. 5 shows the compressed output stream format of the proposed HW. M_i is the metadata

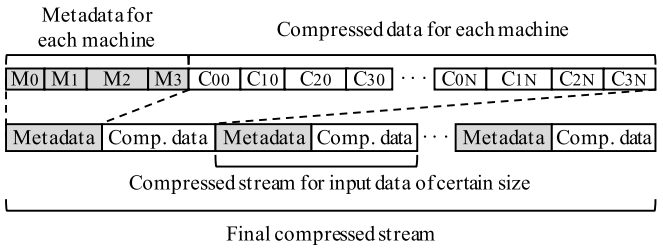


Fig. 5. Compressed output stream format. The certain size is the product of the block size and the number of Huffman machines.

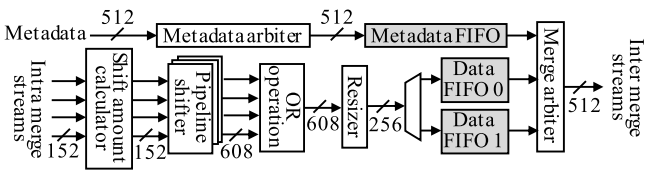


Fig. 6. Architecture of intermerge module in parallel stream merge.

of the ith Huffman machine module. C_{ij} is the compressed data of the jth cycle of the ith Huffman machine after the first output is started. Fig. 6 shows the structure of intermerge module in the proposed parallel stream merge to construct such a compressed stream efficiently. A Huffman codeword is generated every cycle, while the metadata are generated for several cycles depending on the symbol that appears in the specified block size. In other words, a Huffman codeword is generated for a certain cycle, whereas the metadata are generated during a variable cycle according to the number of symbols used in the block. The timing characteristics of the metadata and Huffman codeword must be considered for high-throughput processing. Accordingly, the metadata characteristics of the narrow width-long cycle must be converted to wide width-short cycle. To achieve this, the metadata are stored in the SRAM. Therefore, a total of four metadata can be easily controlled by storing them into one FIFO. These processes are performed through the metadata arbiter and metadata FIFO, as shown in Fig. 6.

For the Huffman codeword processing, actual codewords generated by the Huffman machine module among the fixed width wire must be judged and only the actual codewords are to be merged. As shown in Fig. 6, the shift amount calculator computes each codeword location using the codeword size information, and the pipeline shifter shifts codewords. Then, OR operation merges several codewords. The intramerge module also contains the corresponding logic. The intramerge module in one Huffman machine merges codewords through seven shifters because eights symbols are processed in parallel. In the intermerge module, three shifters are used to merge each intramerged codeword, as four Huffman machines operate in parallel. The merging module including the shifter is fully pipelined for high-throughput processing and can be implemented at a low cost because the HW wire width is considerably reduced by the block unit compression. The resizer reduces the output merge stream once more to fit the FIFO width. Furthermore, the resized data are stored in the

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

data FIFOs. By using the FIFOs, the Huffman codeword path can be easily controlled considering the timing of the metadata path. Two FIFOs are used for the I/O throughput control.

An arbitration module is required to pack the metadata and the Huffman codeword, as shown in Fig. 5. This process is performed through the merge arbiter in the intermerge module of Fig. 6. When metadata are packed into the output stream, the merged Huffman codeword stream must be blocked. After packing four metadata into the output stream, the merged Huffman codeword stream stored in the FIFOs is packed. Finally, the final compressed stream is stored in the buffer inside the stream out module according to the format in Fig. 5. The stream out module outputs the final compressed stream in a specific size.

C. Preprocessing for DDR4 Memory Command Data

1) Don't Care Bits Override: There are various commands [34] depending on the memory operation. In particular, read, write, precharge, and device deselect occupy most of the DDR4 memory command. Among these commands, the device deselect command occupies a considerable portion compared to the other commands because it is necessary to complete the operations that require more than a single clock cycle such as bank active, burst read, and refresh. In the device deselect command, the CKE0, CKE1, $\overline{\text{CS0}}$, $\overline{\text{CS1}}$, $\overline{\text{CS2}}$, and $\overline{\text{CS3}}$ signals among the 32 bits have high (i.e., "1") values and the remaining 26 bits have don't care (i.e., "×") bits. Therefore, 26 don't care bits can be arbitrarily designated to lower the entropy of the trace data because the device deselect command can be judged by only 6 bits (i.e., CKE0, CKE1, $\overline{\text{CS0}}$, $\overline{\text{CS1}}$, $\overline{\text{CS2}}$, and $\overline{\text{CS3}}$ signals). In other words, the compression efficiency can be enhanced by increasing the appearance frequency of a specific symbol by arbitrarily overriding the don't care bits to 0 or 1. It is noteworthy that it is impossible to exclude the don't care bits because the output stream from lossless compression is composed of variable length. If the don't care bits in the device deselect command are excluded in the encoding process, the symbol values of the data are changed and, consequently, the location of the device deselect command is unknown in the decoding process, and it is impossible to judge which commands correspond to the decoded symbol values. Although a flag or bitmap for the location of device deselect commands can be used to solve this problem, this results in a significant additional overhead in terms of HW resource and latency.

To increase the data bias, it is important to efficiently determine a 0 or 1 to the don't care bits in the device deselect command. For the operation of the DDR4 memory, the read, write, and precharge commands occupy most of the entire command. Therefore, the proposed algorithm is applied to bit signals $\overline{\text{ACT}}$, $\overline{\text{RAS}}$, $\overline{\text{CAS}}$, and $\overline{\text{WE}}$, which are fixed to 1 or 0 in these commands (i.e., read, write, and precharge commands) and have don't care bits in the device deselect command. Table I shows the overriding values for the don't care bits of the device deselect command. $\overline{\text{ACT}}$ and $\overline{\text{RAS}}$ signals are overridden by 1 because they are more frequently fixed to 1 in read, write, and precharge commands, whereas $\overline{\text{CAS}}$ and $\overline{\text{WE}}$ signals are overridden by 0 because they are more frequently fixed to 0 in these commands. The compression efficiency can be increased by lowering the data entropy through a preprocessing process that efficiently handles such don't care bits.

TABLE I
DON'T CARE BITS OVERRIDE

Function	$\overline{\text{ACT}}$	$\overline{\text{RAS}}$	$\overline{\text{CAS}}$	$\overline{\text{WE}}$
Read	1	1	0	1
Write	1	1	0	0
Pre-charge	1	0	1	0
Deselect	**1**	**1**	**0**	**0**

2) Bits Arrange: To further enhance the compression efficiency, the correlation of the data to be compressed together in Huffman coding must be increased. To efficiently compress the command data of 32 bits through block Huffman coding, which compresses by 8-bit symbol units, this paper proposes a method to rearrange 32 bits of data utilizing the correlation between each bit and group these data in units of 8 bits to use as the block Huffman coding input. Consequently, these partitioned data have much lower entropy than the original command data, and hence, better compression efficiency can be achieved in block Huffman coding.

For data grouping, mutual information, which is an index indicating the degree of correlation, is used and can be expressed as follows:

$$I(X;Y) = H(X) + H(Y) - H(X,Y) \tag{2}$$

where $H(X)$ and $H(Y)$ are the entropy values of random variables X and Y, respectively; $H(X, Y)$ denotes the joint entropy of the variables set. It is noteworthy that the higher the correlation, the higher is the mutual information value. In the proposed scheme, it is necessary to calculate the mutual information between all bits to divide 32 bits of command data into units of 8 bits and group the bits with high correlation together. However, as the number of possible combinations is large (about 4.15×10^{15}), it is impossible to obtain all mutual information in practice. To efficiently address this problem, this paper applies a heuristic method to obtain the mutual information in a 1-bit unit, and then group the bits with high correlation using the spectral clustering scheme [37] and K-medoids scheme [38], commonly used in machine-learning clustering. If the mutual information is obtained in 1-bit units, mutual information of 32×32 matrix type can be obtained. However, this mutual information that must be tied together for large edges cannot be applied directly to existing clustering methods because existing clustering methods are grouping for low edges; thus, matrix data conversion is necessary. Spectral clustering [37] is an appropriate method to allow mapping of high-affinity data to low values and, consequently, mutual information after spectral clustering can be applied to the existing clustering method. The mutual information data processed by the spectral clustering are nonvector space data, and theoretically, the best representation of the object set in the nonvector space is widely known as a medoid [39]. Also, the number of clusters for "bits arrange" is fixed to four. Moreover, the hard clustering method in which one object

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

TABLE II
BITS ARRANGE

Group	Command bits							
0	A0	A1	A2	AP	A11	$\overline{BC}$	A13	$\overline{CS3}$
1	A3	A4	A5	A6	A7	A8	A9	$\overline{WE}$
2	$\overline{CAS}$	$\overline{RAS}$	A17	CKE0	CKE1	C2	$\overline{CS2}$	$\overline{ACT}$
3	$\overline{CS0}$	$\overline{CS1}$	ODT0	ODT1	BA0	BA1	BG0	BG1

should not belong to several clusters is required for mutual information data. Considering all these features, k-medoids, an improved version of the well-known k-means algorithm [40], is the best clustering algorithm for the data containing mutual information. It is noteworthy that it exhibits excellent clustering performance [41] and has been widely used to obtain optimal results [42], [43]. By applying these machine learning techniques to the memory command data, the optimal clustering result in the data extracted for the experiment can be obtained. Table II shows the results of the bits arrange scheme by clustering. As shown in Fig. 1, when the input data enter the preprocessing module, the entropy is reduced first through the don't care bits override module, and the command bits are divided into four groups, as shown in Table II. The result of this grouping is finally input to the block Huffman coding module and compressed in each module.

IV. EXPERIMENTAL RESULTS

In this section, the HW implementation results and a compression ratio of the proposed system are compared with those of the previous studies to show the superiority of the proposed HW design. To acquire the experimental data set, an intermediate board is placed between the CPU and memory, and the memory command trace data are extracted by hijacking. This paper utilized the DDR4 synchronous dynamic random access memory from SK Hynix [44], which is commercialized and actively used, and the diagnostic application for trace extraction is operated on server-class computers of major vendors. These applications contain most of the trends that memory can experience (i.e., from worst cases to general cases). Furthermore, enough traces (i.e., sufficient capacity) including various characteristics have been extracted. In other words, the workloads used in the experiments reflect most of the memory operation characteristics, and consequently, it can be said that it is sufficient to evaluate the performance of various studies.

A. Experimental Results of the Proposed Design

The proposed HW design is implemented in the Vivado 2017 design tool and operated on the Xilinx Virtex Ultrascale VCU108 field-programmable gate array (FPGA) [45] at the frequency of 250 MHz. The block size for the block Huffman coding in the proposed HW design is 16 kB.

Table III shows the implementation results for each module in Fig. 1. The preprocessing module uses 0.02% of the look-up tables (LUTs) and 0.02% of registers in the configurable logic block among all resources. SRAM is not used. Although this module does not occupy a large part in the whole system,

TABLE III
IMPLEMENTATION RESULTS OF EACH MODULE

Resource		Preprocessing	Block Huffman	Parallel stream merge
Number of CLB LUTs	Available	537,600		
	Utilization	112	42,052	12,208
	%	0.02	7.82	2.27
Number of CLB Registers	Available	1,075,200		
	Utilization	257	24,366	15,753
	%	0.02	2.27	1.47
SRAM (KB)	Available	7782.4		
	Utilization	0	256.13	179.9
	%	0	3.29	2.31

it shows a very high compression efficiency improvement in the memory command trace data. For the block Huffman module, 7.82% of the LUTs and 2.27% of the registers in the FPGA are used. In addition to these resources, it requires 3.29% of the SRAM in the FPGA as it utilizes SRAM-based queues, bitmaps, and code table. This means that block Huffman coding is efficiently implemented using only a small amount of LUTs and registers by utilizing SRAMs appropriately. In a parallel stream merge, the proposed HW structure only utilizes 2.27% of LUTs, 1.47% of registers, and 2.31% of SRAM with the reduction in wire width using the mathematical properties of Huffman coding, data processing by FIFO, and bit size resizing. In addition, no timing failure occurred in the proposed HW design. It is noteworthy that when processing variable length streams for a 32 B/cycle input with the conventional Huffman coding [16], 8192 bits of wires are required and, consequently, the usages of LUTs, registers, and SRAM are 13.75%, 14.75%, and 11.4%, respectively, which causes very large HW cost. In this case, negative slack is also generated by the shifter for the merge and the logic involved in selecting the metadata and Huffman codeword. These results show that the proposed HW architecture enables an efficient parallel stream merge processing at a very low cost. In summary, the proposed HW architecture utilizes 11.38% of LUTs, 4.76% of registers, and 5.6% of SRAM. All modules do not use DSP. At 250 MHz, the worst negative slack is 0.004 ns, the worst hold slack is 0.03 ns, and no total negative slack and total hold slack were present.

Table IV shows the entropy results of the original data (i.e., 512 MB) and the preprocessed data (i.e., 128 MB each). The entropy is an indicator of how much data can be represented as small information. It is noteworthy that the smaller the entropy of the data, the higher is the compression ratio. In general, the memory trace data are organized according to a complicated standard called JEDEC [34], and therefore, the pattern distribution of the trace data is irregular and it has a high entropy. The original entropy of the experimental data has a high value of 0.827 on average owing to the various characteristics of the DDR4 memory command trace data; therefore, it is impossible to obtain high compression ratio. However, using the two preprocessing techniques proposed in Section III-C, the entropy can be reduced significantly. The average entropy after "don't care bits override" and "bits arrange" are 0.610 and 0.607, respectively. By using both methods together, the average entropy can be reduced to

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

TABLE IV
EXPERIMENTAL RESULTS IN TERMS OF THE ENTROPY

Data	Original	Don't care bits override	Bits arrange				Don't care bits override & Bits arrange			
			Split group							
			0	1	2	3	0	1	2	3
1	0.804	0.554	0.319	0.896	0.381	0.811	0.187	0.453	0.307	0.432
2	0.835	0.640	0.379	0.835	0.324	0.870	0.254	0.528	0.304	0.568
3	0.829	0.618	0.358	0.861	0.349	0.866	0.230	0.512	0.308	0.540
4	0.831	0.617	0.359	0.869	0.323	0.866	0.231	0.514	0.293	0.539
5	0.835	0.620	0.378	0.870	0.353	0.866	0.242	0.515	0.311	0.540
Average	0.827	0.610	0.359	0.867	0.346	0.856	0.229	0.504	0.305	0.524

TABLE V
EXPERIMENTAL RESULTS FOR THE COMPRESSED SIZE

Data	Without preprocessing	With preprocessing				
	Compressed size (MB)	Compressed size (MB)				Total size (MB)
		Group 0	Group 1	Group 2	Group 3	
1	382.20	23.68	53.62	31.32	48.65	157.26
2	439.60	38.02	71.30	40.44	74.51	224.27
3	435.38	35.60	69.45	40.14	70.63	215.83
4	437.43	35.56	70.10	38.76	70.57	214.99
5	437.56	35.32	69.69	39.36	70.52	214.89
Average	426.43	33.64	66.83	38.00	66.98	205.45

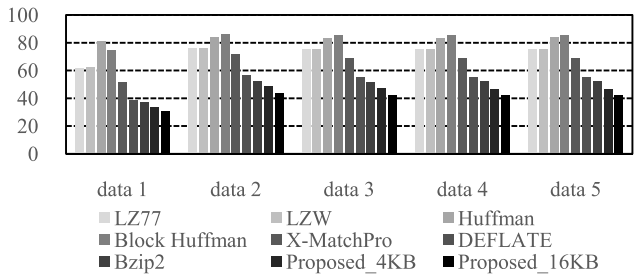


Fig. 7. Comparison results of the compression ratio.

0.391 for the four groups; consequently, a sufficiently high compression ratio can be achieved using the block Huffman coding. Table V shows the compressed size results of the proposed method. The results show that the block Huffman coding of 16-kB block size has a low compression ratio of 83.29% on average in raw data without preprocessing, but has a high compression ratio of 40.13% on average after preprocessing. In addition, the standard deviation between the compressed results is also quite low. As a result, the proposed design can obtain high compression ratio without being greatly affected by the characteristics of the input data because it adaptively forms the Huffman tree according to the input data and utilizes two preprocessing methods that reflect the operating characteristics of the DDR4 memory. It is noteworthy that this paper proposes the novel preprocessing schemes focusing on the DDR4 memory command trace data, but various preprocessing modules can be used with the block Huffman coding depending on the target application.

B. Performance Comparison

To show the superior compression efficiency of the proposed scheme, Fig. 7 shows the compression ratios of LZ77 [12], Lempel–Ziv–Welch (LZW) [15], Huffman [16], block Huffman [26], X-MatchPro [21], DEFLATE [17], Bzip2 [20], and the proposed method. For a fair compression ratio comparison, the block size of the proposed compression method is set to 4 kB, which is the smallest block/dictionary size of the previous compression algorithms. It is noteworthy that the larger the block size, the higher is the compression ratio. In addition, as the proposed HW architecture is designed to process block sizes up to 16 kB to enhance the compression efficiency, the compression ratio results using the block size of 16 kB are also presented to show the accurate performance of the proposed HW design. The average compression ratios of the LZ77, LZW, Huffman coding, block Huffman coding, X-MatchPro, DEFLATE, and Bzip2 are 72.59%, 72.97%, 82.93%, 83.29%, 65.68%, 52.05%, and 49.12%, respectively. However, even with a 4-kB block size, the average compression ratio of the proposed method is 44.49%, which is higher than all other methods. If the block size is 16 kB, the proposed method can obtain an average compression ratio of 40.13%. Although Bzip2, which is the combination of several transforms, run-length encoding, and Huffman coding, has the highest compression ratio among the previous methods, the proposed method shows better compression ratio than Bzip2 despite the lower complexity.

Table VI shows the HW performance of the proposed HW accelerator and other lossless compression HW designs. It is noteworthy that the final target of the proposed HW design is FPGA, not application specified integrated circuit, and the previous studies also presented the LUT results based on the FPGA implementation. Therefore, the usages of LUTs and SRAMs are used to compare the HW complexity of each design presented in the seventh and eighth rows, respectively. In case of the power consumption in the ninth row, the power consumption in logics is simulated on design compiler with the Taiwan Semiconductor Manufacturing Company 65-nm process and that in SRAMs is estimated based on the previous study [46]. The HW complexity of the previous studies is predicted based on the previous studies [21], [23], [47], [48]. The ratio of LUT usage of compressor and decompressor is approximately 3:2 [47] and doubling the dictionary size increases chip complexity by a factor of 1.5 [21]. Therefore, the LUT of the compressor used by Nunez and Jones [21] is estimated by applying these ratios to the total LUT results in [21]. Next, the LUT of the compressor used by Lin and Chang [23] is estimated using a power dissipation of compression and decompression, which is approximately

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

TABLE VI
COMPARISON RESULTS OF HARDWARE PERFORMANCE

	Nunez *et al*. [21]	Lin *et al*. [23]	Fowers *et al*. [24]	Proposed
Algorithm	X-MatchPro	PDLZW+AHDB	DEFLATE	Block Huffman
FPGA	Xilinx Virtex E	Xilinx Spartan 3	Altera Stratix V	Xilinx Virtex Ultrascale
Dictionary or Block Size	128 B	296 B	64 KB	16 KB
Input width	4 Bytes	1 - 5 Bytes	32 Bytes	32 Bytes
Clock speed	50 MHz	68 MHz	175 MHz	250 MHz
Complexity *	4,830 LUTs	2,083 LUTs	195,030 LUTs	62,791 LUTs
SRAM size *	3.19 KB	7.88 KB	192 KB	436.03 KB
Total power *	11.14 mW	4.86 mW	450.14 mW	148.79 mW
Normalized HW resource (Kbits)	188.25	133.22	8106.83	5603.76
Throughput	200 MB/s	113 MB/s	5.6 GB/s	8 GB/s
Power/Throughput	0.056	0.043	0.078	0.018
Normalized HW/Throughput	0.941	1.179	1.414	0.684

* Estimated value

1:1 ratio [23]. It should be noted that there is a proportional relationship between the power dissipation and HW resources of the modules which operate in the same operating environment [49]. In case of Fowers *et al.* [24], the LUT result of the compressor is estimated from the adaptive logic module (ALM) results in [24] based on the ratio that the ALM has a density advantage of approximately 1.8 times the LUT [48].

For a fair comparison, the total HW cost including LUTs and SRAMs is required. Because the LUTs and SRAMs have different units, it is impossible to simply conduct a fair comparison including both LUTs and SRAMs. To address this difficulty, the tenth row of Table VI compares each study by using the HW resources normalized by the number of transistors in each LUT and SRAM. According to the transistor-level design [50], [51], one LUT can be replaced with a 34.5-bit SRAM to estimate the normalized HW resource. Finally, as a tradeoff relationship exists between HW resource and throughput, the performance is compared by a normalized parameter, the total HW resource per throughput, for a fair comparison with previous studies, as shown in the last row of Table VI. It is noteworthy that the HW design with the smaller values denotes better HW designs because it requires less HW resources to achieve the same performance. The results of Nunez and Jones [21], Lin and Chang [23], and Fowers *et al.* [24] show 0.941, 1.179, and 1.414, respectively, while the proposed HW has a very low value of 0.684. As a result, the proposed HW architecture has the highest compression efficiency of 40.13% and the best HW resource per throughput performance compared to the previous studies. These results shown in Fig. 7 and Table VI demonstrate the excellence of the proposed HW design.

V. CONCLUSION

As memory command trace analysis becomes significant in improving memory performance, memory command trace compression is becoming increasingly important. In particular, to support the rapid increase in the amount of data, the demand for a high-throughput HW architecture of lossless compression is increasing considerably. The two main contributions of this paper are as follows. First, a low-cost HW design of lossless compression, which maintains the high compression efficiency and high throughput, is proposed. Experimental results demonstrate that the proposed HW design has a much better total HW resource per throughput than the previous studies. Second, to further enhance the compression ratio for the memory command trace data, two preprocessing methods, the "don't care bits override" and the "bits arrange," which utilize the operating characteristics of the DDR4 memory, are proposed. Experimental results demonstrate that the proposed schemes can achieve an excellent compression ratio of 40.13% on average while maintaining a high throughput of 8 GB/s. It is noteworthy that although this paper applies preprocessing to the DDR4 memory command data, the proposed Huffman HW design has a high scalability, which can be extended to various applications by different preprocessing according to the target data. Therefore, the proposed design is expected to contribute greatly to the reduction in large amounts of data.

REFERENCES

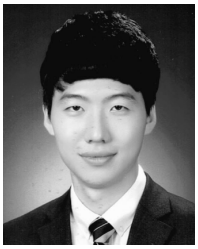
[1] H. Choi, J. Lee, and W. Sung, "Memory access pattern-aware DRAM performance model for multi-core systems," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw.*, Apr. 2011, pp. 66–75.
[2] S. Rixner, W. J. Dally, U. J. Kapasi, P. R. Mattson, and J. D. Owens, "Memory access scheduling," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, May 2000, pp. 128–138.
[3] Y. Huang *et al.*, "HMTT: A hybrid hardware/software tracing system for bridging the DRAM access trace's semantic gap," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 1, Feb. 2014, Art. no. 7.
[4] D.-I. Jeon, M.-K. Lee, J.-C. Kim, and K.-S. Chung, "Runtime memory controller profiling with performance analysis for DRAM memory controllers," *J. Circuits, Syst. Comput.*, vol. 27, no. 8, p. 1850126, Nov. 2017.
[5] C. F. Kao, S. M. Huang, and I. J. Huang, "A hardware approach to real-time program trace compression for embedded processors," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 54, no. 3, pp. 530–543, Mar. 2007.
[6] M. Burtscher, I. Ganusov, S. J. Jackson, J. Ke, P. Ratanaworabhan, and N. B. Sam, "The VPC trace-compression algorithms," *IEEE Trans. Comput.*, vol. 54, no. 11, pp. 1329–1344, Nov. 2005.
[7] K.-U. Irrgang and T. B. Preußer, "An LZ77-style bit-level compression for trace data compaction," in *Proc. 25th Int. Conf. Field Program. Logic Appl.*, Sep. 2015, pp. 1–4.
[8] V. Uzelac and A. Milenkovic, "A real-time program trace compressor utilizing double move-to-front method," in *Proc. 46th ACM/IEEE Design Automat. Conf.*, Nov. 2009, pp. 738–743.
[9] A. Milenkovic, V. Uzelac, M. Milenkovic, and M. Burtscher, "Caches and predictors for real-time, unobtrusive, and cost-effective program tracing in embedded systems," *IEEE Trans. Comput.*, vol. 60, no. 7, pp. 992–1005, Jul. 2011.
[10] *High Bandwidth Memory DRAM*, JEDEC Standard JESD235A, 2015.

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

[11] B. Sukhwani, B. Abali, B. Brezzo, and S. Asaad, "High-throughput, lossless data compresion on FPGAs," in *Proc. IEEE 19th Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2011, pp. 113–116.
[12] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.
[13] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Trans. Inf. Theory*, vol. IT-24, no. 5, pp. 530–536, Sep. 1978.
[14] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *J. ACM*, vol. 29, no. 4, pp. 928–951, Oct. 1982.
[15] T. A. Welch, "A technique for high-performance data compression," *IEEE Comput.*, vol. C-17, no. 6, pp. 8–19, Jun. 1984.
[16] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.
[17] L. P. Deutsch, *DEFLATE Compressed Data Format Specification Version 1.3*, document RFC 1951, May 1996.
[18] M. Burrows and D. Wheeler, "A block sorting lossless data compression algorithm," Digit. Equip. Corp., Maynard, MA, USA, Tech. Rep. 124, 1994.
[19] J. L. Bentley, D. D. Sleator, R. E. Tarjan, and V. K. Wei, "A locally adaptive data compression scheme," *Commun. ACM*, vol. 29, no. 4, pp. 320–330, Apr. 1986.
[20] J. Seward. (2010). *Bzip2 Version 1.0.6*. [Online]. Available: http://www.bzip.org
[21] J. L. Nunez and S. Jones, "Gbit/s lossless data compression hardware," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 3, pp. 499–510, Jun. 2003.
[22] M. B. Lin, J. F. Lee, and G. E. Jan, "A lossless data compression and decompression algorithm and its hardware architecture," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 9, pp. 925–936, Sep. 2006.
[23] M. B. Lin and Y. Y. Chang, "A new architecture of a two-stage lossless data compression and decompression algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 9, pp. 1297–1303, Sep. 2009.
[24] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on FPGAs," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, pp. 52–59.
[25] D. E. Knuth, "Dynamic Huffman coding," *J. Algorithms*, vol. 6, no. 2, pp. 163–180, Jun. 1985.
[26] M. A. Mannan and M. Kaykobad, "Block Huffman coding," *Comput. Math. Appl.*, vol. 46, nos. 10–11, pp. 1581–1587, Nov. 2003.
[27] Y. S. Abu-Mostafa and R. J. McEliece, "Maximal codeword lengths in Huffman codes," *Comput. Math. Appl.*, vol. 39, no. 11, pp. 129–134, Oct. 2000.
[28] A. Jas, J. Ghosh-Dastidar, M.-E. Ng, and N. A. Touba, "An efficient test vector compression scheme using selective Huffman coding," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 22, no. 6, pp. 797–806, Jun. 2003.
[29] J. H. Pujar and L. M. Kadlaskar, "A new lossless method of image compression and decompression using Huffman coding techniques," *J. Theor. Appl. Inf. Technol.*, vol. 15, no. 1, pp. 18–23, 2010.
[30] S. Shanmugasundaram and R. Lourdusamy, "A comparative study of text compression algorithms," *Int. J. Wisdom Based Comput.*, vol. 1, no. 3, pp. 68–76, Dec. 2011.
[31] S. Porwal, Y. Chaudhary, J. Joshi, and M. Jain, "Data compression methodologies for lossless data and comparison between algorithms," *Int. J. Eng. Sci. Innov. Technol.*, vol. 2, no. 2, pp. 142–147, Mar. 2013.
[32] R. A. Bedruz and A. R. F. Quiros, "Comparison of Huffman algorithm and Lempel-Ziv algorithm for audio, image and text compression," in *Proc. Int. Conf. Humanoid, Nanotechnol., Inf. Technol., Commun. Control, Environ. Manage. (HNICEM)*, Dec. 2015, pp. 1–6.
[33] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic coding for data compression," *Commun. ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987.
[34] *DDR4 SDRAM*, JEDEC Standard JESD79-4A, 2013.
[35] J. V. Leeuwen, "On the construction of Huffman trees," in *Proc. 3rd Int. Colloq. Automata, Lang. Program.*, Jul. 1976, pp. 382–410.
[36] S. Rigler, W. Bishop, and A. Kennings, "FPGA-based lossless data compression using Huffman and LZ77 algorithms," in *Proc. IEEE Can. Conf. Elect. Comput. Eng.*, Apr. 2007, pp. 1235–1238.
[37] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Proc. Adv. Neural Inf. Process. Syst.*, 2001, pp. 849–856.
[38] J. Kaufman and P. J. Rousseeuw, "Clustering by means of medoids," in *Statistical Data Analysis Based on the L1-Norm and Related Methods*, Y. Dodge, Ed. Amsterdam, The Netherlands: North Holland, 1987, pp. 405–416.
[39] J. Zhou and J. Sander, "Data bubbles for non-vector data: Speeding-up hierarchical clustering in arbitrary metric spaces," in *Proc. 29th Int. Conf. Very Large Data Bases*, Sep. 2003, pp. 452–463.
[40] J. A. Hartigan and M. A. Wong, "Algorithm AS 136: A K-means clustering algorithm," *Appl. Stat.*, vol. 28, no. 1, pp. 100–108, 1979.
[41] H.-S. Park, J.-S. Lee, and C.-H. Jun, "A K-means-like algorithm for K-medoids clustering and its performance," in *Proc. Int. Conf. Comput. Inf. Eng.*, Jan. 2006, pp. 102–117.
[42] N. Arbin, N. Suhaimi, N. Mokhtar, and Z. Othman, "Comparative analysis between K-means and K-medoids for statistical clustering," in *Proc. 3rd Int. Conf. Artif. Intell., Modelling Simulation (AIMS)*, Dec. 2015, pp. 117–121.
[43] Y. Hamzaoui, M. Amnai, A. Choukri, and Y. Fakhri, "Novel clustering method based on K-medoids and mobility metric," *Int. J. Interact. Multimedia Artif. Intell.*, vol. 5, no. 1, pp. 29–33, Jun. 2018.
[44] *DDR4 SDRAM Device Operation*, SK Hynix Inc., Icheon, Korea, 2018.
[45] *VCU108 Evaluation Board User Guide*, Xilinx Inc., San Jose, CA, USA, 2017.
[46] E. Morifuji, D. Patil, M. Horowitz, and Y. Nishi, "Power optimization for SRAM and its scaling," *IEEE Trans. Electron Devices*, vol. 54, no. 4, pp. 715–722, Apr. 2007.
[47] S. S. Naqvi, R. Naqvi, R. A. Riaz, and F. Siddiqui, "Optimized RTL design and implementation of LZW algorithm for high bandwidth applications," *Electr. Rev.*, Jan. 2011, pp. 279–285.
[48] Altera Corp., San Jose, CA, USA. *FPGA Architecture WP-01003-1.0*. Accessed: Jun. 20, 2018. [Online]. Available: https://www.altera.com/en_US/pdfs/literature/wp/wp-01003.pdf
[49] C. Cernazanu-Glavan, S. Fedeac, A. Amaricai-Boncalo, and M. Marcu, "Energy profiling of FPGA designs," in *Proc. IEEE Int. Symp. Robot. Sens. Environ.*, Oct. 2014, pp. 118–123.
[50] N. Azizi and F. N. Najm, "Look-up table leakage reduction for FPGAs," in *Proc. IEEE Custom Integr. Circuits Conf.*, Sep. 2005, pp. 186–189.
[51] T. Pi and P. J. Crotty, "FPGA lookup table with transmission gate structure for reliable low-voltage operation," U.S. Patent 6 667 635 B1, Dec. 23, 2003.

Jiwoong Choi received the B.S. degree in electrical and electronics engineering from Chung-Ang University, Seoul, South Korea, in 2015 and the M.S. degree in electrical and computer engineering from Seoul National University, Seoul, in 2017, where he is currently working toward the Ph.D. degree in electrical and computer engineering.

His current research interests include hardware accelerator design, computer architecture, and SoC design.



Boyeal Kim received the B.S. degree in electrical and computer engineering from Seoul National University, Seoul, South Korea, in 2017, where he is currently working toward the integrated M.S. and Ph.D. degrees in electrical and computer engineering.

His current research interests include high-bandwidth memory, computer architecture, and hardware accelerator design.

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.

Hyun Kim (M'12) received the B.S., M.S. and Ph.D. degrees in electrical engineering and computer science from Seoul National University, Seoul, South Korea, in 2009, 2011, and 2015, respectively.

From 2015 to 2018, he was a BK Assistant Professor at the BK21 Creative Research Engineer Development for IT, Seoul National University. In 2018, he joined the Department of Electrical and Information Engineering, Seoul National University of Science and Technology, Seoul, where he is currently an Assistant Professor. His current research interests include algorithm, computer architecture, memory, and SoC design for low-complexity multimedia applications, and deep neural networks.

Hyuk-Jae Lee (M'04) received the B.S. and M.S. degrees in electronics engineering from Seoul National University, Seoul, South Korea, in 1987 and 1989, respectively, and the Ph.D. degree in electrical and computer Engineering from Purdue University, West Lafayette, IN, USA, in 1996.

From 1998 to 2001, he was a Senior Component Design Engineer at the Server and Workstation Chipset Division, Intel Corporation, Hillsboro, OR, USA. From 1996 to 1998, he was a Faculty Member at the Department of Computer Science, Louisiana Tech University, Ruston, LS, USA. In 2001, he joined the School of Electrical Engineering and Computer Science, Seoul National University, where he is currently a Professor. He is the Founder of Mamurian Design, Inc., Seoul, a fabless SoC design house for multimedia applications. His current research interests include computer architecture and SoC design for multimedia applications.

Authorized licensed use limited to: Nanjing University. Downloaded on October 14,2020 at 06:59:47 UTC from IEEE Xplore. Restrictions apply.