Received September 26, 2019, accepted October 10, 2019, date of publication October 14, 2019, date of current version October 25, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2947273

Design of FPGA-Based LZ77 Compressor With Runtime Configurable Compression Ratio and Throughput

SEUNGDO CHOI[1], YOUNGIL KIM[1], DAEYONG LEE[1], SANGJIN LEE[1], KIBIN PARK[1], (Student Member, IEEE), YUN HEUB SONG[1], AND YONG HO SONG[1,2], (Member, IEEE)

[1]Department of Electronics and Computer Engineering, Hanyang University, Seoul 04763, South Korea

[2]Samsung Electronics Company Ltd., Hwaseong 18448, South Korea

Corresponding author: Yong Ho Song (yhsong@hanyang.ac.kr)

This work was supported in part by the Ministry of Trade, Industry and Energy (MOTIE) under Grant 10080613, and in part by the Korea Semiconductor Research Consortium (KSRC) support program for the development of the future semiconductor device.

ABSTRACT Data compression reduces the cost of data storage and transmission by decreasing the data size. Previous studies have improved system performance by adaptively choosing the compression ratio (CR) and throughput required for the system by using a trade-off between them in the compression algorithm. Hardware accelerators are widely used to reduce the CPU load caused by compression operations. Several existing compression accelerators have low flexibility in changing the CR and bandwidth. This study proposes a hardware compression accelerator that can adjust the CR and throughput at runtime. The proposed architecture accelerates the LZ77 compression algorithm and supports the throughput-first (TF) and compression ratio-first (CF) modes by changing the degree of parallelism of comparison operations performed during the compression process. In addition, we propose a technique to dynamically change the degree of parallelism of the comparison operation to achieve a better throughput in CF mode and a better CR in TF mode. Experimental results demonstrate that the TF mode provides a throughput higher by 11.39%, and a CR lower by 0.07 than the CF mode. The value 0.07 accounts for 13.21% of the variation in the CR provided by the software implementation of LZ77.

INDEX TERMS Accelerator architectures, data compression, field programmable gate arrays.

I. INTRODUCTION

Recently, the volume of digital data has been continuously increasing owing to the development of information communication technology [1], [2]. State-of-the-art technologies such as the Internet of Things (IoT), social networks, and artificial intelligence (AI) are further accelerating the data growth. With the increase in the volume of digital data, the cost of data storage and transmission has also increased. To reduce these costs, several studies have researched data compression [3]–[7]. Data compression reduces the data size, helps improve storage efficiency, and reduces the bandwidth required for transmission.

The associate editor coordinating the review of this manuscript and approving it for publication was Junxiu Liu.

Compression algorithms can be evaluated on two major indicators. The first is CR, which is calculated as original data size divided by compressed data size. CR indicates how much data size is reduced by the compression algorithm. The second is throughput, which indicates how fast the original data is compressed.

Generally, compression algorithms have a trade-off relationship between CR and throughput [8]–[11]. These algorithms can achieve a higher CR if the time and space complexities are increased. Conversely, sacrificing CR can reduce the time and space complexities of these algorithms, which results in an increase in throughput.

LZ77 is one of the compression algorithms that replaces redundant *string*, i.e., a set of characters of arbitrary length with short length code [9]. The algorithm performs a number of comparison operations to search the repeated string, and the CR and throughput vary according to the number of comparison operations.

Previous studies have pointed out that the appropriate CR and throughput differ according to the context of the computing system [12]–[15]. These studies use a trade-off between CR and throughput to adaptively select the appropriate CR and throughput for the system context under consideration, thereby increasing the service capacity of the web server or reducing the I/O response time of the storage.

Similar to the other tasks performed on a computing system, compression also requires computing resources such as CPU and memory. Compression can be a system overhead if the compression workload required by the system is considerable or a high CR is required.

The hardware acceleration technique is a method used to reduce the computational load of a system. This technique migrates some or all of the tasks that the CPU has performed to a hardware device such as a GPU [16], an ASIC [17], or an FPGA [17]–[23]. Some previous researches studied compression acceleration techniques and achieved high bandwidth by parallelizing frequently performed operations [20]–[22].

Previously proposed compression accelerators provide higher bandwidth than a CPU. However, previous structures have low flexibility in modifying CR and bandwidth. It may be difficult to apply compression that is appropriate to the system context or application characteristics.

This study proposes a hardware compression accelerator that can appropriately prioritize CR and throughput. The proposed architecture adjusts the CR and throughput by changing the degree of parallelism of the string search performed during the compression process and the comparison operation required for the string search. In other words, this architecture increases the degree of parallelism of the string search and decreases the degree of parallelism of the comparison operation to provide high throughput and low CR (*throughput-first mode; TF mode*) or vice versa to provide high CR and low throughput (*compression ratio-first mode; CF mode*). The operation mode of the accelerator is selectable every time a new compression input is assigned, and does not require hardware change.

The proposed accelerator determines the degree of parallelism of the comparison operation dynamically. To determine the degree of parallelism during compression, the proposed architecture uses a comparison validity determination technique [24] to determine the level of parallelism that does not affect the CR. The necessity of additional adjustment of the degree of parallelism is determined according to the current operation mode of the compression accelerator and whether the necessary comparison operation can be performed in parallel within one cycle.

We evaluated the proposed architecture using FPGA. The proposed accelerator provides throughput of 1.76 bytes/cycle and CR of 2.48 in TF mode and throughput of 1.58 bytes/cycle and CR of 2.55 in CF mode. Compared to the CF mode, in the TF mode, the throughput increased by 11.39%; however, the CR decreased by 0.07. This accounts for approximately 13.21% of the variation of the CR provided by the software implementation of LZ77 library.

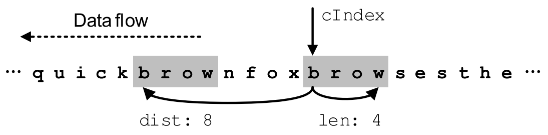

FIGURE 1. Example of LZ77 operation.

LZ77 is also used as part of a compression library (e.g., deflate [25]). We implemented and evaluated the deflate accelerator using the proposed architecture. The deflate accelerator achieved a throughput and CR of 3.16 GB/s and 3.42, respectively, in CF mode and a throughput and CR of 3.52 GB/s and 3.37, respectively, in TF mode.

The rest of this paper is organized as follows. In section II, we present background knowledge of this study and in section III, we briefly introduce the related work. Section IV provides an overview of the proposed architecture and section V explains the method to make the CR and throughput selectable in the accelerator. Section VI discusses the experimental results of the architecture. Finally, we conclude this paper in section VII.

II. BACKGROUND

A. LZ77 OVERVIEW

LZ77 is a lossless data compression algorithm. Lossless data compression does not allow for data loss during compression; thus, the decompression result is identical to the original data. LZ77 reduces the data size by replacing redundant strings with shorter length codes. It comprises four steps: target string selection for searching, string search, substitution decision, and outputting results. These four steps are performed sequentially in a circular manner. In this study, we refer to the flow from the first to the last stage as *unit operation*.

Fig. 1 shows an example of the LZ77 operation. *cIndex* refers to the starting position of the target string (or *current string*), and it is an index that distinguishes between the part of the data that was compressed (before cIndex) and the part to be compressed (starting at cIndex). The cIndex moves each time a unit operation is performed. LZ77 compares the current string with previous strings to identify if the current string is duplicated. In the example, the string "brow" starting from cIndex appeared in duplicate. LZ77 replaces the current string with a pair set (*LD pair*) comprising the matched length of the two strings (*len*) and the distance between the two strings (*dist*), and the cIndex shifts by the length of the substituted string. If the current string is not duplicated, the character indicated by cIndex (*literal*) is output. In this case, cIndex moves to the next character. The unit operation of the LZ77 is repeated until cIndex reaches the end of the data.

B. STRING SEARCH USING HASH TABLE

The zlib [8] is a compression library used by several applications [26]. It is an abstraction of the deflate algorithm and includes LZ77 and Huffman encoding [27].

LZ77 implemented in zlib performs comparison operations on strings of three-byte length. If a duplicate string exists, the target and the previously appeared string are further checked to see if they match more than three bytes, and the matched string is then replaced with the LD pair. The maximum length of a matched string in zlib is set to 258.

Comparing all three-byte strings with the target string consumes considerable time. To reduce the search time, zlib uses a hash table to manage the information of starting positions of previous strings (*history*). zlib uses the target string as the key of the hash table and stores the starting position of the target string in the hash bucket. Further, using a target string as key, zlib can obtain the starting position of the strings with the same key value. Instead of comparing all strings in the compressed part, zlib improves the time efficiency of the search process by only comparing strings starting from histories stored in the corresponding hash bucket.

C. REDUCING COMPARISON PROCESS WITH FALSE HISTORY FILTERING

Hash table uses a hash function to obtain the access address of a bucket. Hash collision is one of the characteristics of a hash function, which implies that different inputs of the hash function produce the same result. In zlib, histories of different strings are stored as linked lists in the same bucket when a hash collision occurs. As a result, even if histories are stored in the same bucket, several strings starting from some of these histories may be different from the target string. Comparing different strings has no effect on the CR. It is not necessary to perform *meaningless*, or *unnecessary* comparison operations. Especially in a hardware compression accelerator, where the number of string comparators (*SCs*) is limited, other meaningful comparisons are delayed considerably if an unnecessary comparison process occupies the comparator. Therefore, eliminating unnecessary comparisons before they are assigned to the comparator helps to yield higher performance.

False history filtering is a technique to detect unnecessary comparison operations that occur during the LZ77 operation [24]. To determine whether histories from the hash table affect CR, this technique stores a portion of the three-byte string (*filtering tag*) used for the hash key with history. The size of the filtering tag ranges from 0 to 24 bits. A size of 0 bits indicates that the hash table does not use the filtering tag, whereas a size of 24 bits indicates that all the three bytes are used for the filtering tag. A compressor using this technique compares the filtering tag with a portion of the target string before assigning the comparison operation to the SC. If the comparison results do not match, the comparison operation is judged not to affect the CR, and it is not assigned to the comparator. False history filtering can reduce the number of comparisons performed in the compression. In this paper, we shall use the terms false history filtering and *history filtering* interchangeably.

III. RELATED WORK

Some existing studies discuss improving the system performance by using compression. Some researches use compression and adaptively adjust the compression level in consideration of idle CPU resources or network congestion for file transfers via networks [13]–[15]. One study suggested adaptive compression techniques to reduce storage response time [12]. The proposed scheme determines whether to use a compression algorithm with low CR and low time cost or one with high CR and high time cost, depending on the expected CR of the file or the number of I/O requests.

Other studies proposed hardware acceleration scheme for compression. Fowers et al. proposed a hardware architecture that accelerates LZ77 and Huffman encoding [27]. The accelerator improves the performance by reducing the number of memory accesses. To reduce memory access time required for linked list traversal in a hash table, the accelerator uses multi-way memory structure such as CPU cache, and stores histories in each way. Each way is divided into several banks to allow multiple accesses to the memory. In the multi-way architecture, all histories stored in the same bucket can be loaded with a single memory access. This not only reduces the number of memory accesses but also enables parallelization of sequential string comparison processes, thereby reducing execution time.

Several accelerator architectures including the abovementioned research: [20]–[22], select multiple strings as target strings in one unit operation, and perform all the comparisons generated in the unit operation in parallel to increase the performance. If more strings are searched simultaneously, more comparison work is required to be done; therefore, these architectures perform parallel comparisons using many comparators. These accelerators achieve high bandwidth; however, users or applications cannot adjust the CR or throughput that they provide.

Our accelerator allows the user to prioritize the CR and throughput without changing the hardware. The architecture changes the degree of parallelism of comparison operation to adjust the CR and throughput. This structure omits some of the comparison operations to change the degree of parallelism, and the number of omitted tasks is determined dynamically.

IV. ARCHITECTURE OVERVIEW

In this section, we describe the proposed LZ77 acceleration architecture. We designed the accelerator based on the prototype of the LZ77 acceleration hardware proposed in the previous study [24]. Unlike the previous structure that searches only one string per unit operation, the proposed architecture selects multiple target strings per one unit operation to improve the performance.

Fig. 2 illustrates the proposed LZ77 accelerator. The operation of this architecture is as follows. First, the data to be compressed is stored in the accelerator memory (data

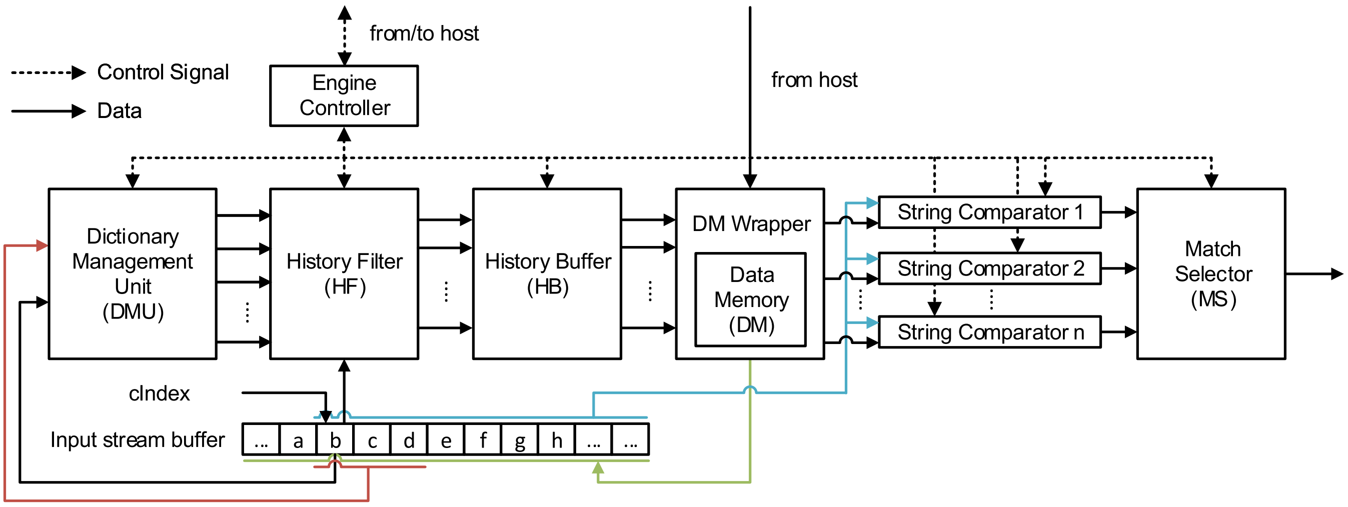

FIGURE 2. **Block diagram of LZ77 accelerator.**

memory, *DM*). The input stream buffer prefetches a portion of the data stored in the DM and provides it as input to the SC. Dictionary management unit (*DMU*) updates and outputs the starting position of the string to be compared by the SC. The starting position of the string is transmitted to the history filter (*HF*), history buffer (*HB*), and DM. DM outputs the data corresponding to the starting position. The string comparator compares the data output from the DM with the string received from the input stream buffer. The comparison results are delivered to the match selector (*MS*), that selects the result with the longest matched length (similar to the case of zlib). Fig. 2 illustrates the data flow when one unit operation is performed. The number of target strings in the figure is set to one for readability.

Details of each module are as follows. Engine controller controls the operation of the accelerator. It receives control signals such as a compression request or an operation mode setting from an external control device (e.g., a host device driver or other controller device), shares it with internal modules, and informs the external device of the completion of compression. DMU performs the same function as that of the hash table of zlib. It records the starting positions of strings that appeared before cIndex. DMU has hash memory with multi-way and multi-bank type [20]. HF is used to reduce comparisons by detecting unnecessary string comparisons [24]. DM stores the data to be compressed. SC compares a string starting from cIndex with a string that has appeared before, determines whether it is redundant, and generates a literal or LD pair based on the result of comparison. MS selects the LD pair with the largest length value among the results generated by SCs. If none of the SCs have generated an LD pair, the MS outputs the literal indicated by cIndex. MS also supports the lazy matching scheme that is implemented in zlib. HB controls the comparison operation according to the operation mode. We describe the detail of the HB in the next section. All modules in the architecture

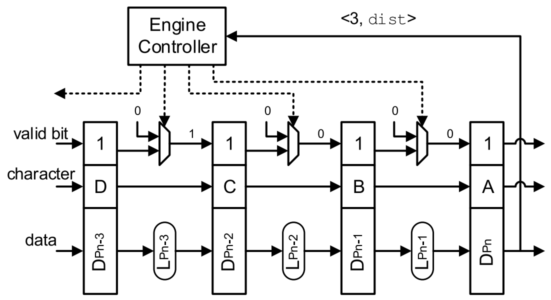

FIGURE 3. **Data flow of LZ77 engine.**

are pipelined, and they can perform new LZ77 unit operations per cycle.

Fig. 3 is a simplified diagram of the pipeline structure and data flow of the proposed architecture, illustrating an example of performing a single string search per unit operation. One unit operation is performed through several stages constituting the accelerator. Each unit operation has the data (D_{px}) necessary to perform the compression, and this data changes according to the logic (L_{px}) in each stage. For example, D_{px} contains the address to access the dictionary when the unit operation is in the DMU, and D_{px} includes the result of the SC when the unit operation is in the MS. Each unit operation also has the character pointed by cIndex, and the character is used as the LZ77 output when target string is not replaced by the LD pair in the unit operation. The valid bit indicates whether the result of the unit operation is to be used as the output of the accelerator. The engine controller receives the compression result from the last stage of MS (last stage in the figure) at every cycle, and it controls the valid bits of the other stages. For example, if an LD pair with an including length value of a is output at the last stage of the MS, the valid bit of the $a-1$ unit operation before the last stage is set to 0, which means invalid. This is because the already compressed part does not need to be output.

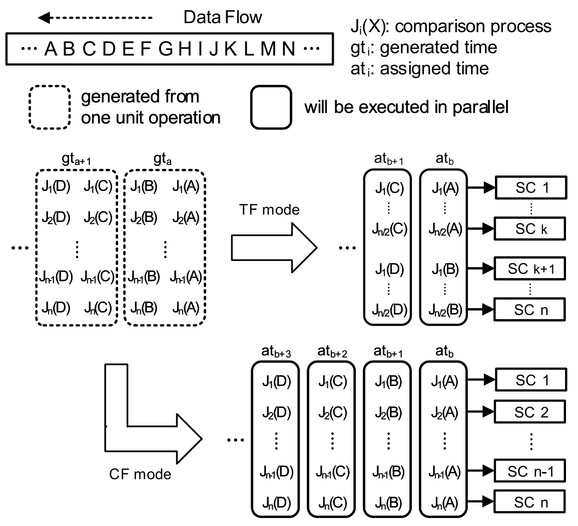

FIGURE 4. **Basic approach for adjusting compression ratio and throughput.**

V. DYNAMICALLY ADJUSTING THE DEGREE OF PARALLELISM

This section explains how to adjust CR and throughput in the proposed architecture. We adjust CR and throughput by changing the degree of parallelism of the string search and comparison operations. To determine the number of comparison operations to be performed, the proposed architecture dynamically determines how many comparison tasks to be omitted to improve throughput and CR in each operation mode.

A. FIRST STEP: LIMIT THE NUMBER OF COMPARISONS

zlib has 9 levels of compression; level 1 provides the fastest performance and lowest CR, whereas level 9 provides the lowest throughput and highest CR. One way to adjust the CR in zlib is to limit the number of comparison operations performed per target string search. For example, level 1 performs a smaller number of comparison tasks than level 9 does. A small number of comparison operations improves performance; however, the probability of not finding a redundant string increases, which can lead to a reduction in CR. On the contrary, a large number of comparison tasks increases the probability of finding duplicated strings and longer redundant strings; however, it costs considerable time.

Fig. 4 shows an example of changing the performance by limiting the number of comparison operations performed per target string search. For the ease of explanation, we assume that the architecture searches two target strings in one cycle, and maximum n comparison operations are generated per one target string. We also assume that the architecture has n comparators ($n = 2k$ and k is a natural number). $J_i(X)$ in Fig. 4 represents the comparison process required for the target string search. For example, $J_i(A)$ is a comparison operation that is generated by searching for the string starting from A in the input data flow. As the value of i increases, it compares a string that is more far from A. If the number of $J_i(X)$ is less than the number of comparators, some comparators remain idle to simplify the internal operation. Comparison tasks created in one unit operation are grouped into rounded rectangles drawn in dashed lines, and the comparison operations to be performed in parallel are grouped into rounded rectangles drawn in solid lines. gt_i indicates the time when the tasks are created; gt_{a+1} is the next cycle of gt_a. at_i is the time when the comparison operations are assigned to the comparators. at_{b+1} is the next cycle of at_b, and the comparison tasks of at_b are assigned to the comparator ahead of at_{b+1}.

Only few of the comparison operations generated at gt_i are performed in TF mode. In Fig. 4, the number of comparison operations that can be performed for one target string is fixed to $n/2$, and distant comparison tasks from the target string are omitted, e.g., the operation of zlib. In the at_b of TF mode, the number of $J_i(A)$ and $J_i(B)$ is $n/2$, and a total of n comparison operations are simultaneously allocated to the comparators (SC). In TF mode, all comparison operations created in gt_i are processed concurrently in at_i. The number of comparison operations generated per cycle is higher than that of the comparators; however, there is no bottleneck because the number of comparison operations actually performed is equal to the number of comparators.

On the contrary, all the comparison operations generated at gt_i are performed without omission in CF mode. In the CF mode example of the figure, all of the operations: $J_i(A)$, $J_i(B)$, $J_i(C)$, and $J_i(D)$ are performed. The number of comparators in the proposed architecture is less than the number of comparison operations generated at gt_a; therefore, the tasks of gt_a should be performed over at_b and at_{b+1}. In CF mode, some comparison operations wait until the comparators become available, and the pipeline stages before the comparator are stalled. Although CF mode yields less throughput than TF mode owing to the pipeline stall, the CR can be increased because it performs more comparison operations.

B. SECOND STEP: DYNAMICALLY DETERMINE THE NUMBER OF COMPARISONS TO BE SKIPPED FOR HIGHER COMPRESSION RATIO AND THROUGHPUT

In the previous section, we discussed the basic method for adjusting the CR and throughput in the proposed architecture. This section describes a technique for improving the performance in CF mode and increasing the CR in TF mode. We further describe the structure and operation of the HB for adjusting the degree of the parallelism of comparison operations.

To improve CR and throughput in each mode, we focus on unnecessary comparisons. Unnecessary comparisons may cause two problems. First, they can limit the CR. In the example of Fig. 4, TF mode omits some comparison operations and performs only some comparison tasks. However, the comparison results may not affect the CR; moreover, the omitted comparisons may lead to a higher CR. Better CR can be expected in TF mode if unnecessary comparisons are detected and not performed before being assigned to the comparator.

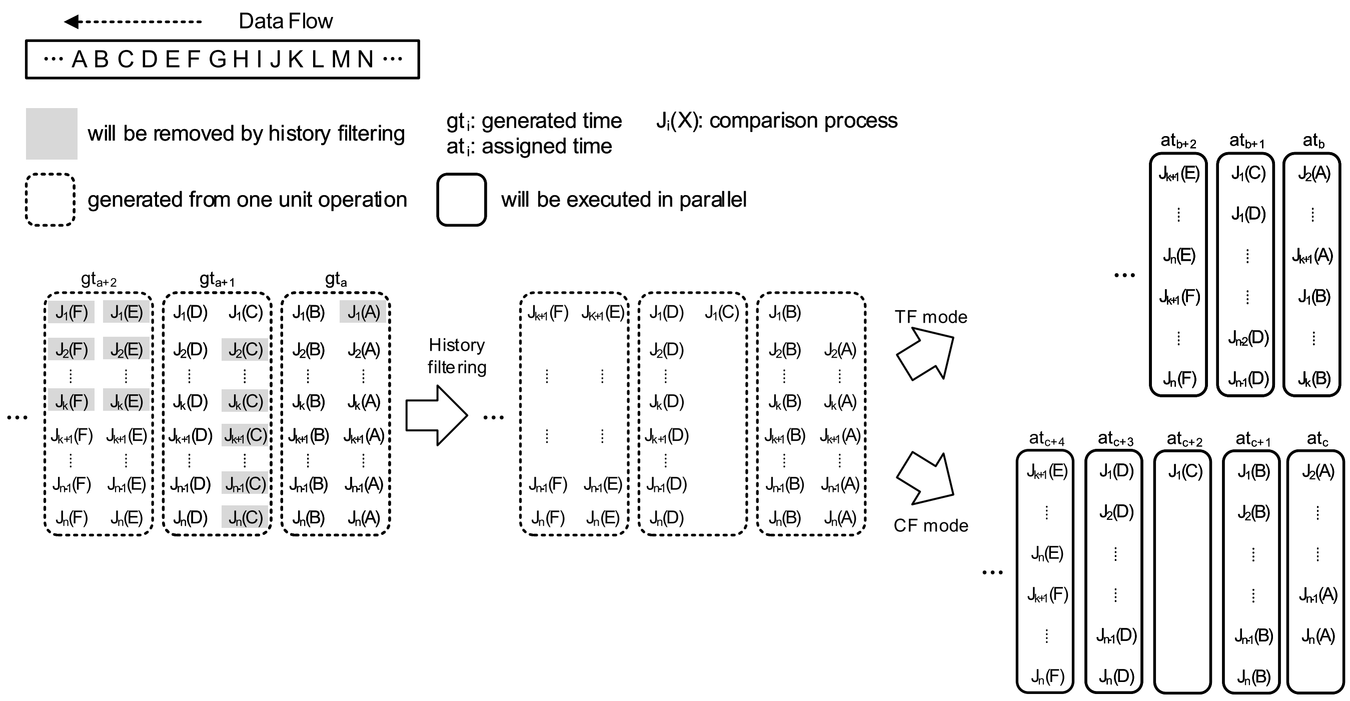
Data Flow
··· A B C D E F G H I J K L M N ···
will be removed by history filtering
generated from one unit operation
gt_i: generated time
at_i: assigned time
J_i(X): comparison process
will be executed in parallel
gt_a+2
gt_a+1
gt_a
History filtering
TF mode
CF mode
at_b+2
at_b+1
at_b
at_c+4
at_c+3
at_c+2
at_c+1
at_c

FIGURE 5. Difference of comparison workload and execution time according to each operation mode with dynamic skip.

Second, unnecessary comparisons may limit performance improvement. We show an example in which the comparison operations generated at gt_i are performed over at_i and at_{i+1} in CF mode of Fig. 4. If half of the comparison operations created at gt_i are unnecessary, and they can be filtered out before being allocated to comparator, the remaining comparison operations can be performed in parallel at at_i.

We use a two-step comparison omission technique that we call *dynamic skip*. The first step in dynamic skip omits unnecessary comparisons using the history filtering scheme [24]. The operation of the second stage is determined according to the current operation mode of the compressor and the number of remaining comparison operations. In TF mode, it is determined whether additional omissions are required, whereas in CF mode it is decided whether to perform the remaining comparison operations in single or multiple cycles. The first step is performed in the HF and the second step in the HB.

Fig. 5 illustrates the operation of each mode in the architecture using dynamic skip. The notations such as gt_i and at_i and the operating assumption in this figure are the same as in Fig. 4. The comparison operations on the gray background are omitted after history filtering.

Both the modes perform history filtering before performing comparison tasks. We assume that $J_1(A)$ generated at gt_a is removed by history filtering. We also suppose that operations from $J_2(C)$ to $J_n(C)$ created at gt_{a+1} would be omitted, and those from $J_1(E)$ to $J_k(E)$ and $J_1(F)$ to $J_k(F)$ generated at gt_{a+2} would be discarded after history filtering. TF mode omits unnecessary comparisons and determines whether additional omission is required, depending on the number of remaining comparison operations. Basically, TF mode with dynamic skip limits the number of $J_i(X)$ to be performed to $n/2$, which is similar to the example in Fig. 4. However, if the number of remaining $J_i(X)$ is less than $n/2$ after discarding unnecessary comparisons, the comparison operations for other strings are allowed to be executed.

In the case of at_b in Fig. 5, $J_i(A)$ is performed, where i takes values from 2 to $k + 1$, which is a total of k. This is owing to the reason that the number of $J_i(A)$ after history filtering is higher than $n/2$. However, $J_{k+1}(A)$, which is omitted in Fig. 4 can be performed instead of $J_1(A)$ because $J_1(A)$ is removed by history filtering. In the case of gt_{a+1}, only one $J_i(C)$ remained. As aforementioned, the dynamic skip allows performing a comparison operation of other strings according to the number of remaining $J_i(X)$. Therefore, $J_i(D)$ at at_{b+1} can be performed by $n - 1$, which is more than $n/2$. Some of $J_i(D)$ cannot be performed if we fix the number of comparisons to be performed to $n/2$, which is similar to that shown in Fig. 4. For gt_{a+2}, unnecessary comparisons are removed, and $J_i(E)$ and $J_i(F)$ remain as $n/2$. In this case, the remaining comparison operations are performed without additional omission. In summary, at_b and at_{b+2} are expected to improve the CR by performing comparisons that may affect the CR instead of unnecessary comparisons. Further, at_{b+1} is expected to improve the CR by omitting less comparison tasks of some target strings.

CF mode does not perform additional omission after filtering to avoid dropping the CR. Instead, it checks whether the remaining compression operations are performed in parallel to increase the throughput. In this mode, comparison operations $J_i(X)$ for one string are processed in the same

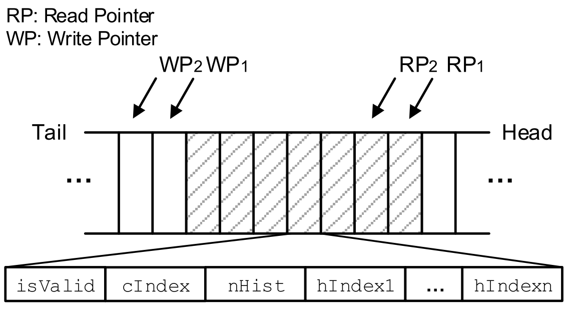

FIGURE 6. hFIFO and its entry.

cycle to simplify the control of the architecture. In case of gt_a and gt_{a+1}, the number of remaining tasks is larger than the number of comparators. Therefore, the remaining operations created at gt_a are performed over at_c and at_{c+1}. Similarly, the remaining operations created at gt_{a+1} are performed at at_{c+2} and at_{c+3}. However, in case of gt_{a+2}, the number of comparison operations remaining after filtering is equal to the number of comparators. The comparison tasks generated at gt_{a+2} had to be performed in multiple cycles if the unnecessary comparisons were not omitted; however, these tasks could be run in parallel on the same cycle at_{c+4} owing to the dynamic skip. In summary, the application of dynamic skip is expected to make the CF mode in Fig. 5 to be faster than CF mode in Fig. 4, in which the number of omissions is fixed to zero. The number of unnecessary comparisons may vary for every cycle, and it is necessary to determine whether parallel processing is possible for each cycle.

HB module either skips comparison operations or checks whether they can be performed in parallel according to the operating mode. This module also stores the information related to the comparison operations for the pipeline control of the situation, where the number of comparison operations to be performed is larger than the number of comparators in CF mode.

Figs. 6 and 7 are concept diagrams of the role and operation of HB. HB comprises history FIFO (hFIFO) that is a FIFO data structure, and DM address selector (DMAS) that controls the inputs of SC and DM. Fig. 6 shows the hFIFO and the data stored in it. This example assumes that two string searches are performed in one cycle.

The number of read pointers (RPs) and write pointers (WPs) in the hFIFO is equal to the number of strings to be checked in one cycle. HB stores the information related to the comparison operation filtered in the HF at the position pointed by the WP and reads the data at the point indicated by the RP at every cycle and transfers it to the DM and the SC. If there is zero available space in the hFIFO, it outputs a 'full' signal, thereby stalling the operation of the pipeline stages located before HB, and waits until the SC completes the comparison operation.

The information stored in hFIFO is as follows. *isValid* indicates whether the result of the comparison operation stored in the current entry is to be used as the output of LZ77; it is the same as the valid bit in Fig. 3. cIndex represents the

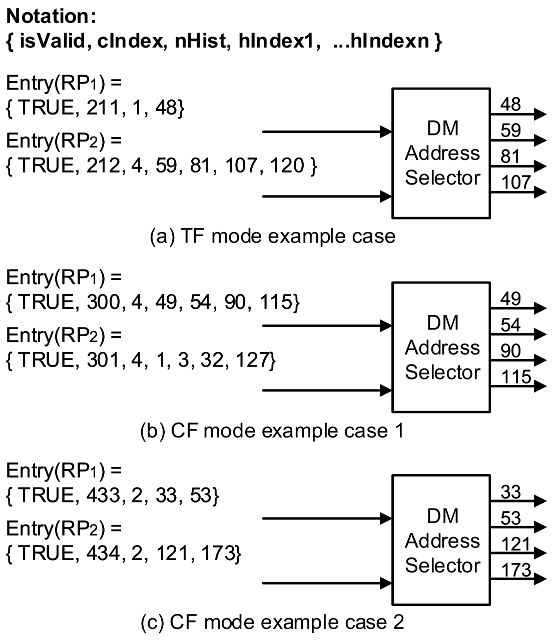

FIGURE 7. Example of address selection.

starting position of the target string. *nHist* is the number of comparison operations remaining after filtering, and *hIndex* indicates the starting position (history) of a string that had appeared previously.

Fig. 7 shows an example of the operation of HB in each operation mode. In this figure, we assume that the architecture has four SCs, two string searches are performed in one cycle, and up to four comparison operations are generated per string. Each example presents a situation in which the entry pointed by RP_1 and RP_2 is output. We refer the nHistory from RP_1 and RP_2 as $RP_1(nH)$ and $RP_2(nH)$, respectively.

Fig. 7 (A) shows the process of omitting comparison operations in TF mode. In 7 (A), the DMAS of HB identifies $RP_1(nH) = 1$ and $RP_2(nH) = 4$; therefore, the DMAS skips the comparison operation of the last history of RP_2 and selects one history from RP_1 and three histories from RP_2 as output. RP_1 and RP_2 move to current position + 2.

Fig. 7 (B) is an example, in which $RP_1(nH) = 4$ and $RP_2(nH) = 4$ in CF mode. The number of total comparison operations is greater than that of the comparators. In this case, the DMAS only selects histories of RP_1 as output. RP_1 and RP_2 move to current position + 1.

Fig. 7 (C) also shows an example of CF mode; however, the number of total comparison operations is equal to that of the comparators ($RP_1(nH) = 2$ and $RP_2(nH) = 2$). All comparison operations can be processed in parallel; therefore, the DMAS outputs four histories from RP_1 and RP_2. RP_1 and RP_2 move to current position + 2.

VI. EXPERIMENTS AND RESULTS

We used Verilog HDL to evaluate the proposed architecture. We implemented the architecture on Xilinx xcku-095 and measured the CR, throughput, and hardware resource usage.

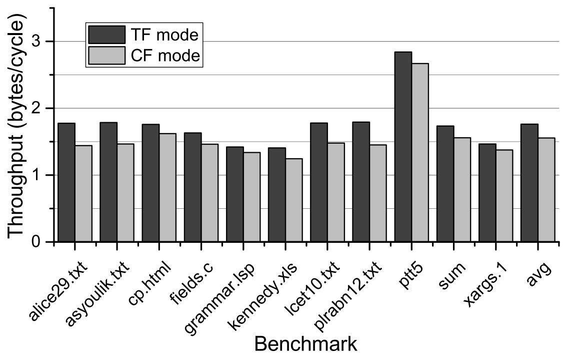

FIGURE 8. Throughput according to operation mode.

We used chunk-based compression that is widely adopted in systems using compression [28], [29]. Chunk-based compression allows hardware accelerators to be implemented with less memory than file-based compression [28]. We also used the Canterbury corpus benchmark suite for evaluation [30].

The size of the hash memory and the DM and the number of bits of filtering tag are from [24]. We organize the hash memory with four ways and 4096 entries per way. By having four ways, we can load up to four histories with one access to the hash memory. This indicates that n (see Fig. 5) is set to four in our experiment. The chunk size is set to 32 KB, which is the same as the size of the zlib internal buffer. The DM has the same memory size as the chunk. We use 7-bit filtering tag, and the HB is set to store 32 entries.

A. THROUGHPUT OF EACH MODE

Throughput indicates the amount of data that can be compressed per cycle. This architecture performs a new unit operation per cycle and reads a certain amount of data per unit operation. If no stall occurs from internal or external module, this architecture can read up to two bytes per cycle.

Fig. 8 shows the throughput according to the operation mode. The x-axis of the graph represents the benchmark files and average, whereas the y-axis represents the throughput of each item. On average, the proposed architecture yields 1.58 bytes/cycle in CF mode and 1.76 bytes/cycle in TF mode. TF mode outperforms CF mode by 11.39%. The average throughput of both modes is slower than two bytes per cycle because of the bank conflict that occurs when the DMU updates the hash memory. If multiple update requests occur on the same bank, they cannot be processed concurrently owing to the lack of memory ports, and must be processed over several cycles. This degrades the throughput of the CF and TF modes.

The ptt5 benchmark shows higher throughput than two bytes per cycle. This is because the lengths of strings that are replaced with LD pair in ptt5 are more. As described in the LZ77 overview, cIndex moves backward by the length of the substituted string, i.e., it does not check for compressibility for the replaced string. The proposed architecture also does not check compressibility for the substituted strings, and consequently reduces the cycle required to compress the data. The average length of the strings replaced by the LD pair in ptt5 was approximately 35, which is higher than the other files.

Throughput of CF mode increases in accordance with the number of times when comparison operations generated in the unit operation are performed within a single cycle, which is affected by the data pattern. As a result, CF mode shows lower throughput than TF mode. In the worst case, the CF mode provides a throughput of approximately 1 byte/cycle; however, the average throughput is higher than that for the benchmark file used in the experiment.

To verify that our proposed design is as meaningful as an accelerator is, we compared the LZ77 throughput of our architecture and that of the CPU. The execution time of LZ77 equals the difference between the time consumed by the deflate function and the time required by Huffman encoding. We compressed the benchmark file with the fastest (zlib level 1) option and measured the execution time of LZ77. The experiment was conducted on Intel Xeon E5-2660 CPU running at 2.6 GHz. The version of zlib used was 1.2.11, and it was compiled with GCC 4.8.5. The average zlib throughput in our experiment was 37.39 MB/s, which approximately equals 0.02 bytes/cycle in terms of throughput per cycle. Experimental results show that the throughput of the proposed accelerator is higher than that of the CPU.

B. COMPRESSION RATIO OF EACH MODE

We measured the LZ77 CR of the proposed architecture. We also measured the LZ77 CR of zlib level 9 to assess the difference between the CR of the proposed structure and the best CR of the software library. We obtain the CR by dividing the size of the original data by the size of the compressed result. In other words, higher CR indicates smaller size of the compressed result.

Table 1 is a summary of CR of zlib and the proposed architecture. The left column lists the benchmark files and average. The following columns represent the size of the original file, the LZ77 CR at zlib level 9, the CR of TF mode, the CR of CF mode, and increment of CR of CF mode compared to TF mode.

The hash memory size of the architecture is lower than that of zlib. As a result, the strings to be compared can be different even in the comparison process for the same target string, and the CR can be reduced [20], [24]. In addition, the CR of chunk-based compression may be lower than that of file-based compression because one chunk does not refer to the dictionary of another chunk. CR of the proposed architecture is lower for most benchmark files when compared to the LZ77 CR of zlib level 9. The CR of TF mode is slightly lower than the LZ77 CR of zlib level 3 (2.49), and the CR of CF mode is equal to the CR of zlib level 4 (2.55).

However, files: grammar.lsp and xargs.1 have higher CR of hardware than zlib, which is owing to the gain of representing the dist of LD pair with fewer bits compared to zlib. The hash memory used can store a total of 16384 histories. We assumed

TABLE 1. Summary of LZ77 CR.

Benchmark file name	file size	CR of zlib level 9	CR of TF mode (①)	CR of CF mode (②)	Difference in CR between operation mode (② - ①)
alice29.txt	152089	2.22	1.86	1.91	0.05
asyoulik.txt	125179	1.99	1.75	1.78	0.03
cp.html	24603	2.42	2.31	2.34	0.03
fields.c	11150	2.67	2.53	2.64	0.11
grammar.lsp	3721	2.17	2.21	2.24	0.03
kennedy.xls	1029744	3.14	2.93	3.07	0.14
lcet10.txt	426754	2.35	1.96	2.00	0.04
plrabn12.txt	481861	1.93	1.63	1.66	0.03
ptt5	513216	7.71	6.35	6.56	0.21
sum	38240	2.35	1.98	2.02	0.04
xargs.1	4227	1.75	1.79	1.81	0.02
average		2.79	2.48	2.55	0.07

TABLE 2. Effect of dynamic skip.

	Throughput (bytes/cycle)		CR	
	with dynamic skip	without dynamic skip	with dynamic skip	without dynamic skip
TF mode	1.76	1.76	2.48	2.42
CF mode	1.58	1.24	2.55	2.56

that recent history with a distance of less than 16384 from the target string is stored in the hash memory. Distance between the target string and the history stored in the hash memory is expressed using 14 bits. The maximum number that can be represented by 14 bits is 16383; therefore, the current string is not replaced by the LD pair if the distance between the current string and history is 16384 or more. On the contrary, zlib allows distance that can be expressed in 15 bits; however, it actually stores the distance using an unsigned short, which is a 16 bit data type. zlib and the proposed architecture both use 8 bits to represent length. Therefore, the proposed architecture uses less bits to represent the LD pair compared to zlib.

In most files, loss of CR owing to smaller hash memory hides the gain from shortened LD pair. However, files: grammar.lsp and xargs.1 show higher CR. If the LD pair is expressed by the same bit-size as in zlib, the CR of the two files in CF mode is reduced to 2.14 and 1.71, respectively, which is lower than that in case of zlib.

Average CR of TF mode is 2.48, whereas that of CF mode is 2.55, which is 0.07 higher than TF mode. The average LZ77 CR of the benchmark in zlib shows a variation of 0.53 from zlib level 1 to level 9, and 0.07 is 13.21% of the variation. TF mode omits comparison operations that may affect the CR, thereby showing a lower CR than CF mode. In ptt5, the number of consecutive appearances of the same character is high; therefore, the length of the string replaced with the LD pair is more. Consequently, its CR is higher than other files.

C. EFFECT OF DYNAMIC SKIP

To verify the effect of dynamic skip, we measured the throughput and CR in each mode without dynamic skip. Table 2 summarizes the effect of dynamic skip. The throughput in TF mode without dynamic skip is 1.76 bytes/cycle, which is the same as the case when dynamic skip is applied. There is no throughput difference because TF mode performs the same number of comparison operations as the number of comparators even if dynamic skip is not applied. However, when dynamic skip is not applied, the CR decreases from 2.48 to 2.42. This is because the filtering in dynamic skip increases the probability of performing comparisons that affect the CR instead of performing unnecessary comparisons.

In CF mode, dynamic skip increased the throughput from 1.24 bytes/cycle to 1.58 bytes/cycle because it causes the comparison operations generated in some unit operations to be processed in a single cycle rather than multiple cycles. Without dynamic skip, the proposed architecture ran approximately at 1 byte/cycle owing to the pipeline stall; however, in practice, higher bandwidth was measured owing to the gain in replacing long strings. The CR when the dynamic skip was not used was 2.56, which showed an increase of 0.01.

History filtering only omits the comparison process that does not affect the CR. The CR changes in the architecture that uses dynamic skip because histories stored in the hash memory may change owing to the throughput difference and not because of history filtering. If dynamic skip changes throughput of the accelerator, the amount of data read by the accelerator varies until a certain point in time. When a long string is substituted, and if the throughput is high, the history of the data portion to be replaced may already have been updated in the hash memory. On the contrary, if the throughput is low, the history of the data portion to be replaced may not be updated in the hash memory. As a result, even if a string starting from the same cIndex is searched, the history from the dictionary may vary depending on whether dynamic skip is applied or not, which may affect the CR. In summary, the architecture using the dynamic skip when compared to that without dynamic skip improves performance in the CF mode, and CR in the TF mode.

D. HARDWARE RESOURCE USAGE

Table 3 summarizes the implementation result of the proposed architecture with Xilinx Vivado 2016.4. Row total represents the total logic elements (LE) [31] and memory units (MU) [32] used in the architecture. Row HF indicates the logic element and memory usage used for history filtering, and row HB represents the logic element and memory usage used for the HB. Each row also shows the proportion of the two resources in the target FPGA.

TABLE 3. Hardware resource usage of proposed architecture.

	LE usage and proportion in target FPGA	MU usage and proportion in target FPGA
Total	11379 (2.12%)	60 (1.79%)
HF	460 (0.09%)	8 (0.24%)
HB	1426 (0.27%)	4 (0.12%)

The proposed architecture uses 11379 logic elements and 60 memory units. In terms of the proportion in the target FPGA, the accelerator consumes approximately 2.12% of the total logic element (537600) and approximately 1.79% of the total memory unit (3360). HF consumes 460 logic elements and 8 memory units. The additional memory to store the filtering tag adopts memory banking, which induces more memory usage than the number mentioned in [24]. HB uses 1426 logic elements and 4 memory units.

E. COMPARISON WITH PREVIOUS STUDIES

To compare our accelerator with other compression accelerators, we implemented a deflate accelerator comprising 16 proposed LZ77 accelerators, 4 Huffman encoders, and 4 bit-packing modules in the target FPGA. Our accelerator is designed to be compatible with the zlib software library, and the output of the deflate accelerator can be decompressed with the library. This accelerator is synthesized without timing violation at 125 MHz and achieves throughput of 3.16 GB/s and 3.52 GB/s in CF and TF modes, respectively.

Table 4 compares the throughput and CR of our accelerator with previous researches. CR (D) and CR (S) denote the CR to which the dynamic and static Huffman encoding is applied, respectively.

TABLE 4. Comparison of throughput and compression ratio with previous studies

	Throughput	CR (D)	CR (S)
Abdelfattah et al. [21]	2.84 GB/s	2.43	–
Fowers et al. [20]	5.60 GB/s	–	2.70
Proposed arch. (CF mode)	3.16 GB/s	3.42	2.73
Proposed arch. (TF mode)	3.52 GB/s	3.37	2.67

Compared to the accelerator proposed in [21], our architecture provides higher throughput and CR. Research [21] focuses on reducing design time and implementation complexity using high level synthesis (HLS), sacrificing CR and prioritizing the performance. Although this architecture provides higher throughput than the CPU, the values of CR and throughput are not adjustable.

Fowers et al. proposed an accelerator that provides scalability for bandwidth [20]. The proposed accelerator achieves maximum bandwidth of 5.60 GB/s and yields CR of 2.70 using static Huffman encoding. This CR is lower than 2.73, which is the CF mode result of our architecture, which is estimated to the effect of limiting the maximum length of the string replacement.

Lee et al. proposed a hardware architecture for accelerating the LZ4 compression algorithm [17]. They achieved a throughput of 0.5 GB/s and a CR of 2.69 using their own test files. Lee et al. stated that their proposed architecture can improve the CR by increasing the amount of input data per cycle; however, this leads to a reduction in the scalability of the hardware.

The performance of the accelerator proposed in [20] is superior to that of ours. However, there is no trade-off between CR and throughput in it. Moreover, architectures proposed in [17] and [20] require a hardware reimplementing process to adjust the performance and CR. Reimplementation and hardware changes may cause system suspension. This makes it difficult to reflect the changes of the system context occurring during runtime and may degrade the performance of the system.

On the other hand, our architecture can adjust the CR and throughput by changing the operation mode at runtime. The accelerator receives a mode-selection signal from the external control device and operates in the mode that corresponds to this signal. This architecture receives the compression start signal and the mode-selection signal at the same time; thus, there is no timing overhead for mode switching. It is expected that the conventional trade-off-based adaptive compression scheme can be adopted even if a hardware compression accelerator is used, thereby improving system performance.

VII. CONCLUSION

In this work, we propose a LZ77 compression accelerator that can adjust the CR and throughput without changing hardware. This architecture operates in CF mode or TF mode by adjusting the degree of parallelism of comparison operations.

We also propose a dynamic skip scheme, which dynamically adjusts the degree of parallelism of comparison operations. Dynamic skip omits unnecessary comparisons in the first step. The operation of the second step depends on the compression mode of the accelerator. When operating in TF mode, it is determined whether additional omission is required for the comparison operation. In CF mode, it is decided whether to operate the remaining comparison operation in single or multiple cycles. The proposed architecture using dynamic skip achieves better throughput in CF mode and higher CR in TF mode compared to the architecture without dynamic skip.

Experimental results using FPGA implementation show that TF mode operates at 1.76 bytes/cycle that is 11.39% faster than CF mode. CR of TF mode is 0.07 lower than CF mode, and is about 13.21% of the maximum variation range provided by software implementation of LZ77.

The deflate accelerator using the proposed architecture yields throughput of 3.16 GB/s and CR of 3.42 CR in CF mode, and throughput of 3.52 GB/s and CR of 3.37 CR in TF mode.

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their valuable feedback and comments.

REFERENCES

[1] J. Gantz and D. Reinsel, "The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east," Int. Data Corp., Framingham, MA, USA, Tech. Rep., Dec. 2012, pp. 1–16.

[2] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, and M. Ayyash, "Internet of Things: A survey on enabling technologies, protocols, and applications," *IEEE Commun. Surveys Tuts.*, vol. 17, no. 4, pp. 2347–2376, 4th Quart., 2015.

[3] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning trained quantization and Huffman coding," in *Proc. Int. Conf. Learn. Represent.*, 2016, pp. 1–14.

[4] A. Zuck, S. Toledo, D. Sotnikov, and D. Harnik, "Compression and SSDs: Where and How?" in *Proc. INFLOW*, Bloomfield, CO, USA, 2014, pp. 1–10.

[5] M. Poess and D. Potapov, "Data compression in Oracle," in *Proc. VLDB*, Berlin, Germany, 2013, pp. 937–947.

[6] G. Pekhimenko, V. Seshadri, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Base-delta-immediate compression: Practical data compression for on-chip caches," in *Proc. PACT*, Minneapolis, MN, USA, 2012, pp. 377–388.

[7] B. Nicolae, "High throughput data-compression for cloud storage," in *Proc. 3rd Int. Conf. Data Manage. Grid P2P Syst.*, Bilbao, Spain, 2010, pp. 1–12.

[8] *Zlib*. Accessed: Jul. 12, 2019. [Online]. Available: http://zlib.net

[9] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. 23, no. 3, pp. 337–343, May 1977.

[10] *Brotli Compression Format*. Accessed: Jul. 12, 2019. [Online]. Available: https://github.com/google/brotli

[11] G. J. Sullivan, J.-R. Ohm, W.-J. Han, and T. Wiegand, "Overview of the high efficiency video coding (HEVC) standard," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 22, no. 12, pp. 1649–1668, Dec. 2012.

[12] B. Mao, S. Wu, H. Jiang, Y. Yang, and Z. Xi, "EDC: Improving the performance and space efficiency of flash-based storage systems with elastic data compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 6, pp. 1261–1274, Jun. 2018.

[13] E. Zohar and Y. Cassuto, "Automatic and dynamic configuration of data compression for Web servers," in *Proc. LISA*, Seattle, WA, USA, 2014, pp. 106–117.

[14] C. Krintz and S. Sucu, "Adaptive on-the-fly compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 17, no. 1, pp. 15–24, Jan. 2006.

[15] E. Jeannot, B. Knutsson, and M. Bjorkman, "Adaptive online data compression," in *Proc. HPDC*, Edinburgh, U.K., Jul. 2002, pp. 371–380.

[16] N. Jacob and C. Brodley, "Offloading IDS computation to the GPU," in *Proc. ACSAC*, Miami Beach, FL, USA, Dec. 2006, pp. 371–380.

[17] S. M. Lee, J. H. Jang, J. H. Oh, J. K. Kim, and S. E. Lee, "Design of hardware accelerator for Lempel-Ziv 4 (LZ4) compression," *IEICE Electron. Express*, vol. 15, no. 11, pp. 1–6, Jun. 2017.

[18] A. Ebrahimi and M. Zandsalimy, "Evaluation of FPGA hardware as a new approach for accelerating the numerical solution of CFD problems," *IEEE Access*, vol. 5, pp. 9717–9727, 2017.

[19] A. Shawahna, S. M. Sait, and A. El-Maleh, "FPGA-based accelerators of deep learning networks for learning and classification: A review," *IEEE Access*, vol. 7, pp. 7823–7859, 2019.

[20] J. Fowers, J.-Y. Kim, D. Burger, and S. Hauck, "A scalable high-bandwidth architecture for lossless compression on FPGAs," in *Proc. FCCM*, Vancouver, BC, Canada, May 2015, pp. 52–59.

[21] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL," in *Proc. IWOCL*, Bristol, U.K., 2014, Art. no. 4.

[22] A. Martin, D. Jamsek, and K. Agarwal, "FPGA-based application acceleration: Case study with GZIP compression/decompression streaming engine," presented at the Special Section 7C ICCAD, 2013.

[23] R. Kobayashi and K. Kise, "A high performance FPGA-based sorting accelerator with a data compression mechanism," *IEICE Trans. Inf. Syst.*, vol. E100.D, no. 5, pp. 1003–1015, May 2017.

[24] S. Choi, Y. Kim, and Y. H. Song, "False history filtering for reducing hardware overhead of FPGA-based LZ77 compressor," *J. Syst. Archit.*, vol. 88, pp. 110–119, Aug. 2018.

[25] *DEFLATE Compressed Data Format Specification Version 1.3*. Accessed: Jul. 12, 2019. [Online]. Available: https://tools.ietf.org/html/rfc1951

[26] *Zlib Applications*. Accessed: Jul. 12, 2019. [Online]. Available: http://zlib.net/apps.html

[27] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. Inst. Radio Eng.*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.

[28] S. Lee, J. Park, K. Fleming, Arvind, and J. Kim, "Improving performance and lifetime of solid-state drives using hardware-accelerated compression," *IEEE Trans. Consum. Electron.*, vol. 57, no. 4, pp. 1732–1739, Nov. 2011.

[29] X. Zhang, J. Li, H. Wang, D. Xiong, J. Qu, H. Shin, J. P. Kim, and T. Zhang, "Realizing transparent OS/Apps compression in mobile devices at zero latency overhead," *IEEE Trans. Comput.*, vol. 66, no. 7, pp. 1188–1199, Jul. 2017.

[30] *The Canterbury Corpus*. Accessed: Jul. 12, 2019. [Online]. Available: http://corpus.canterbury.ac.nz/descriptions/#cantrbry

[31] *7 Series FPGAs Configurable Logic Block*. Accessed: Jul. 12, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[32] *Block Memory Generator V8.3—LogiCORE IP Product Guide*. Accessed: Jul. 12, 2019. [Online]. Available: https://www.xilinx.com/support/documentation/ip_documentation/blk_mem_gen/v8_3/pg058-blk-mem-gen.pdf

SEUNGDO CHOI received the bachelor's and master's degrees in electronics and computer engineering from Hanyang University, Seoul, South Korea, in 2012 and 2014, respectively, where he is currently pursuing the Ph.D. degree in electronics and computer engineering.

His research interests include high-performance computing, computer architecture, and low-power systems.

YOUNGIL KIM received the bachelor's degree in media communication engineering from Hanyang University, Seoul, South Korea, in 2012, where he is currently pursuing the Ph.D. degree in electronics and computer engineering.

His research interests include high-performance computing, lossless data compression, and 3D integrated circuit.

DAEYONG LEE received the B.S. degree from the School of Electronic Engineering, Soongsil University, Seoul, South Korea, in 2014, and the master's degree with the Department of Electronics and Computer Engineering from Hanyang University, Seoul, in 2017, where he is currently pursuing the Ph.D. degree with the Department of Electronical Engineering.

His research interests include embedded systems and NAND flash memories.

SANGJIN LEE received the bachelor's degree in electronics and computer engineering from Hanyang University, South Korea, in 2014, where he is currently pursuing the Ph.D. degree with the Department of Electronic and Computer Engineering.

His research interests include storage systems based on non-volatile memory, system architecture, and host interface.

KIBIN PARK received the bachelor's degree from the Department of Computer Science and Engineering from Hanyang University, in 2015, where he is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering.

His research interests include non-volatile memories, embedded systems, and hardware acceleration.

YUN HEUB SONG received the M.S. degree in electronic engineering from Hanyang University, Seoul, South Korea, in 1992, and the Ph.D. degree in intelligent mechanical engineering from Tohoku University, Sendai, Japan, in 1999.

He is currently a Professor of Electronic Engineering with Hanyang University. He has researched semiconductor devices and circuit design for over 30 years in the Semiconductor Research and Development Center, Samsung Electronics Company, and Hanyang University. When he was working with Samsung, he was responsible for the device and product development of flash memory as the Vice-President, and developed 256 Mb and 512 Mb NOR flash memory, from 2000 to 2003. After moving to Hanyang University in 2008, where he served as the Vice Dean of the College of Engineering, involving in extensive international collaboration research and planning on industrial projects, from 2011 to 2013.

His research interests include device reliability modeling, device characterization, novel device structures and architectures for memory and logic applications, circuit design and algorithms for low-power, high-speed processing, and sensor systems based on semiconductor technology.

YONG HO SONG received the bachelor's and master's degrees in computer engineering from Seoul National University, Seoul, South Korea, and the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, CA, USA, in 1989, 1991, and 2002, respectively.

He is currently a Professor with the Department of Electronic Engineering, Hanyang University, Seoul, and a Senior Vice President of Samsung Electronics Company Ltd. His current research interests include system architecture and software systems of mobile embedded systems that further include SoC, NoC, multimedia on multicore parallel architecture, and NAND flash-based storage systems.

Prof. Song has served as a Program Committee Member of several prestigious conferences, including the IEEE International Parallel and Distributed Processing Symposium, the IEEE International Conference on Parallel and Distributed Systems, and the IEEE International Conference on Computing, Communication, and Networks.