IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 15, 2020, accepted February 8, 2020, date of publication February 24, 2020, date of current version March 2, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2975929

Low-Overhead Compressibility Prediction for High-Performance Lossless Data Compression

YOUNGIL KIM[1], SEUNGDO CHOI[1], DAEYONG LEE[1], JOONYONG JEONG[1], JAEWOOK KWAK[1], JUNGKEOL LEE[1], GYEONGYONG LEE[1], SANGJIN LEE[1], KIBIN PARK[1], JINWOO JEONG[1], WANG KEXIN[1], (Student Member, IEEE), AND YONG HO SONG[1, 2], (Member, IEEE)

[1]Department of Electronics and Computer Engineering, Hanyang University, Seoul 15588, 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 Research and Development Program of MOTIE/KEIT (Developing Processor-Memory-Storage Integrated Architecture for Low Power, High Performance Big Data Servers) under Grant 10077609.

ABSTRACT As big data has evolved over the past few years, a lack of storage space and I/O bandwidth has become one of the most important challenges to overcome. To mitigate these problems, data compression schemes reduce the amount of data to be stored and transmitted at the cost of additional CPU overhead. Many researchers have attempted to reduce the computational load imposed on the CPU by data compression using specialized hardware. However, space savings through data compression often comes from only a small portion of data. Therefore, compressing all data, regardless of data compressibility, can waste computational resources. Our work aims to decrease the cost of data compression by introducing a selective data compression scheme based on data compressibility prediction. The proposed compressibility prediction method provides more fine-grained selectivity for combinational compression. Additionally, our method reduces the amount of resources consumed by the compressibility predictor, enabling selective compression at a low cost. To verify the proposed scheme, we implemented a DEFLATE compression system on a field-programmable gate array platform. Experimental results demonstrate that the proposed scheme improves compression throughput by 34.15% with a negligible decrease in compression ratio.

INDEX TERMS Data compression, Huffman coding, LZ77 encoding, accelerator architecture, field programmable gate array, estimation, compressibility.

I. INTRODUCTION

Recently, as mobile devices as well as information and communication technology have evolved, the amount of data to be stored and processed has increased explosively [1]–[4]. In particular, new IT technologies such as social network services and the internet of things contribute much to the increase in data production. To handle this explosive data growth, large-scale data processing methods, such as MapReduce [5], [6] and cloud computing [7] have been developed. Such applications are very data intensive, requiring a large amount of storage space and huge I/O bandwidth capacity [8]–[11]. Based on these extreme requirements, systems with insufficient storage and computing resources can suffer from serious performance degradation. Therefore, it is important to minimize the required storage space and I/O bandwidth utilization.

Data compression schemes reduce storage space and I/O bandwidth utilization requirements in exchange for increased computational overhead for the CPU. The concept of data compression has been widely applied in many applications to save storage space and transmission costs for networks [12], [14]. In recent years, cloud storage and enterprise data centers have also utilized data compression to reduce data management costs [6], [15]–[26]. According to one study [17], compression phase constitutes a large portion of the overall execution time of warehouse-scale computing.

There are two main types of compression: lossy and lossless. Lossy compression reduces the size of data by sacrificing certain information that is difficult for humans to perceive. Therefore, it is mainly used for compressing files that are still usable with a small amount of lost data, such

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

as multimedia files. In contrast, lossless compression guarantees the integrity of decompressed data. Such methods have worse compression efficiency and higher complexity than lossy methods. However, it has the advantage of the complete restoration of the original data. Therefore, lossless compression is used when the minimization of information loss is critical, such as in text files.

Lossless compression is divided into three types: dictionary-based compression, statistical compression, and combinational compression [27]. Combinational compression comprises two or more compression algorithms: usually a combination of dictionary-based and statistical compression [12], [27]. DEFLATE [28] is one of the most widely used combinational lossless compression algorithms.

The compression quality and time complexity of data compression algorithms have a trade-off relationship. As the complexity of a data compression algorithm increases to improve data compression quality [29], [30], it takes a longer time to process data. Additionally, high-quality real-time data compression requires several algorithms to operate simultaneously [12], [27], [31]–[33], which increases the computational demand on the processor. Although multicore processors are generally used, the execution of data compression algorithms still imposes a major load on systems. This is because most systems cannot handle other tasks while compression algorithms are running.

For this reason, there have been many attempts to reduce the computational load of data compression by utilizing customized hardware resources, such as field-programmable gate arrays (FPGAs) [34]–[41], graphics processing units, and application-specific integrated circuits [42], [43]. These methods decrease the computational load on the processor by offloading a portion of the computations from the processor to a hardware offloading engine. Furthermore, the offloading engine accelerates offloaded computations through parallel processing.

Additionally, to reduce the computational demand, selective compression is used, which selectively compresses according to the compressibility of the data. All data does not compress equally. According to one study [44], 86% of compression savings comes from 50% of file-based server data, indicating that there is a significant skew in the data compressibility distribution. Compressing incompressible data can waste computational resources without any additional space savings. Therefore, it is inefficient to compress all data chunks while ignoring their compressibility. Selective data compression [12] based on the prediction of data compressibility allows a compression system to efficiently utilize limited computational resources.

Many previous studies have investigated techniques to predict the compressibility of data chunks and improve the performance of data compression based on prediction results [12], [13], [22], [45]–[47]. However, these previous studies have two limitations. First, previous studies do not fully account for changes in the system in which compression works. In storage systems, compression is typically performed on fixed-size chunks of data, for a variety of reasons, such as read amplification, metadata overhead, and IO parallelism [48], [49]. In this case, the compression efficiency of the preceding compression algorithm can affect the compression efficiency of subsequent compression algorithms. Secondly, the existing methods do not consider the hardware resource usage. Previous studies mainly focus on software-centric compressibility prediction methods. They do not address the hardware implementation of compressibility prediction. There are always limited hardware resources available to implement processing units. FPGAs have limited hardware resources available within reconfigurable devices. Therefore, to leverage the full potential of the limited hardware resource, the amount of hardware resources used to handle the same task should be minimized.

In this paper, we propose a DEFLATE compression offloading engine using a selective data compression method based on data compressibility prediction. The proposed method analyzes input data chunks in an online manner to predict data compressibility. Based on the compressibility prediction results, the compression engine determines the compression mode for each input data chunk. The contributions of this paper are as follows:

- Our heuristic-based compressibility prediction technique provides an enhanced evaluation of the compression efficiency of chunked-based compression. To accurately estimate the compression efficiency of Huffman coding, our method considers the input chunk size and the compression ratio of the LZ77 additionally. As a result, incompressible data chunks for Huffman coding can be identified more accurately, which reduces the computation of the compressor.
- The proposed hardware cost reduction technique reuses the computation of existing compression algorithms in implementing compressibility prediction. It also reduces the compressibility of computation for compressibility prediction. These techniques significantly reduce the overhead for the compressor to operate selectively, increasing the work efficiency of the compressor.
- To evaluate the effectiveness of the proposed technique, we implement a pipelined compression offload engine for the DEFLATE algorithm. The implemented offload is equipped with the proposed compressibility prediction method. The offload engine is synthesized for an FPGA platform. The performance is measured by running various benchmark applications.

The proposed scheme minimizes performance degradation caused by the failure to predict data compressibility via parallelization with data compression. Furthermore, we propose a method to reduce hardware costs using statistical indicators, such as shift operations and absolute values calculations, which can be implemented using simple hardware logic.

To evaluate the effectiveness of the proposed technique, we designed a DEFLATE compression engine equipped with the proposed compressibility prediction method. The performance of the proposed technique was measured on an

FPGA platform. Experimental results demonstrate that the prediction scheme achieves a 34.15% increase in throughput by sacrificing only 0.09% of the compression rate for incompressible data. According to synthetic experimental results, compressibility prediction uses approximately 3.1% more lookup tables (LUTs) compared to a compressor without such capability.

The remainder of this paper is organized as follows. Section 2 briefly introduces the concepts of DEFLATE compression and information entropy. Section 3 describes related works. The motivation for the proposed compressibility prediction scheme is detailed in section 4. Section 5 outlines features based on the analysis of DEFLATE compression and the proposed compressibility prediction scheme. Section 6 discusses the hardware implementation of the proposed method. Experimental results are presented in section 7. Section 8 concludes this work.

II. BACKGROUND

A. DEFLATE COMPRESSION ALGORITHM

DEFLATE [28] is a well-known lossless compression algorithm that was first used in Zip and Gzip software. DEFLATE utilizes Huffman coding and modified LZ77 encoding. It encodes data using LZ77 encoding and dynamic Huffman codes generated based on collected statistics. One of three compression modes is selected for each input data chunk. The remainder of the compression process is performed according to the selected compression mode. Fig.1 presents a flow diagram of the DEFLATE compression algorithm.

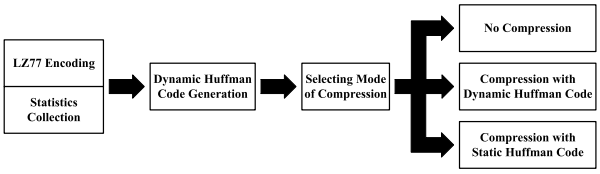


FIGURE 1. Flow diagram of the DEFLATE algorithm.

1) LZ77 ENCODING

The LZ77 algorithm is a dictionary-based compression algorithm [50]. This algorithm searches for repeated strings in input data and replaces them with a pointer to the preceding string. Fig. 2 illustrates the LZ77 encoding process. Because the first "right" does not have any previous appearance, the original characters (denoted as *literal*) are emitted with no modification. Next, the algorithm identifies the second "right" and replaces it with a pointer to the preceding "right". The pointer represents the relative position (denoted as *distance*) of the preceding string and the length of the string (denoted as *length*). The DEFLATE algorithm collects statistical information for Huffman code generation during LZ77 encoding. This process is referred to as *statistics collection*. The statistical information includes the number of unique symbols and appearance frequency of each symbol.

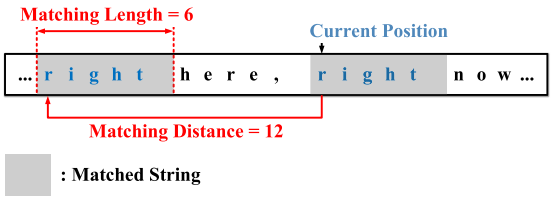


FIGURE 2. Example of LZ77 encoding.

2) HUFFMAN CODING

Huffman coding [51] is a type of entropy coding that compresses data by replacing symbols with variable-sized Huffman codes. Symbols with higher frequencies are assigned relatively shorter Huffman codes compared to symbols with lower frequencies.

Huffman coding is divided into two types: dynamic and static. Static coding uses predefined Huffman codes and compresses data using the same Huffman table, regardless of the content of the data to be compressed. Dynamic coding generates Huffman codes based on the occurrence frequency of each symbol included in the raw data. It builds dynamic Huffman trees with symbols as leaf nodes. Fig. 3 presents an example of a Huffman tree. As shown in the figure, the leaf nodes with higher frequency values are located closer to the root node and are assigned relatively shorter codes.

Dynamic coding has the advantage of generating Huffman codes that are optimized for the current input data. However, it has a disadvantage in that the dynamic Huffman tree must be transmitted with the encoded data stream. Therefore, if the compression efficiency of the dynamic code is poor, the encoded data can be larger than the original data. Additionally, because the dynamic codes must be generated independently for certain amounts of data, the data processing speed is slower than that of static coding.

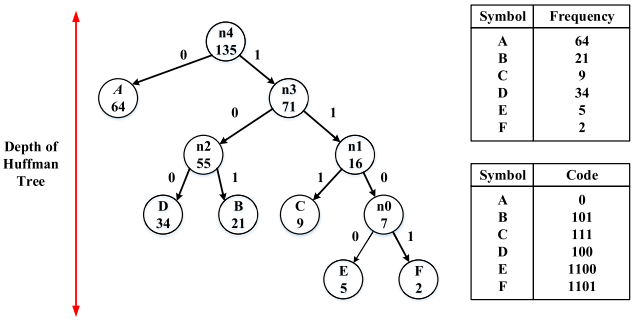


FIGURE 3. Dynamic Huffman tree build by Huffman algorithm.

3) MODES OF COMPRESSION

The DEFLATE algorithm runs on fixed amounts of input data called *data block (DB)*. DEFLATE has three different compression modes (denoted as *DEFLATE mode*). Each DB can be processed in three different ways [52]. The first mode, which is called *stored block mode (SBM)*, simply divides data into DBs with no compression. This mode is suitable for data with low compressibility because such data represents

compressed or encrypted files. In the second mode, which is called *static Huffman mode (SHM)*, data is compressed via LZ77 encoding and static Huffman coding. This mode can be used when the efficiency of dynamic Huffman coding is poor or when data must be compressed at high speed, such as in real-time processing. The third mode, which is called *dynamic Huffman mode (DHM)*, compresses data via LZ77 encoding and dynamic Huffman coding. This mode is typically slower than the other modes but has greater compression efficiency.

To select the best mode with the best compression efficiency, the algorithm compares the length of each compressed DB when each DEFLATE mode is used after the dynamic Huffman code is generated. When DHM is used, the compressed data length is calculated with the size of the dynamic Huffman tree included.

B. INFORMATION ENTROPY

Information entropy is the average amount of information related to an event, which indicates the uncertainty of the event [53]. The formula for information entropy is as follows.

$$H(x) = -\sum_{i=1}^{n} p(i) \log_2 p(i) \tag{1}$$

The higher the entropy of a symbol, the higher the uncertainty of the symbol and the greater the amount of information the symbol represents.

From the viewpoint of information entropy, files can be divided into two types [54]. The first type is a well-compressible text file. In the case of text files, such as documents, web pages, and logs, most streams consist of alphanumeric characters. Therefore, these files are easy to predict which characters will appear in the data stream because the number of characters that can appear is relatively small. These files have low entropy. The second type is a binary file that does not compress well. Binary files, such as images, video, and sound files, have random data patters. Therefore, these files make it difficult to predict which characters will appear in the data stream and have high entropy values.

III. RELATED WORK

A. Kattan et al. predicted the compression ratio of data based on genetic programming [45]. Their method measures compression ratios very accurately, but requires significant time to construct that decision tree that is required to predict the compression ratios. Our approach differs from this method in that we predict compression ratios online using several features that are computationally simple.

Culhane et al. presented three indicators that can measure compression efficiency and changes in the compression efficiency of several compression algorithms based on the three indicators [46]. However, the authors presented only the indicators and provided no discussion regarding how to utilize them for efficient data compression.

Some studies have attempted to compress data selectively or adaptively based on the compressibility of each data chunk. One study measured data compressibility to determine whether or not to compress data chunks stored in the cloud [22]. The author estimated the compressibility of an entire data chunk by compressing a portion of the data chunk and measuring its compressibility. However, this method has the risk of incorrectly measuring the compression rate when the compressibility of a data chunk fluctuates. We use the entire data chunk to accurately measure compressibility, regardless of fluctuations in compressibility. To reduce the implementation overhead, our compressibility estimation process utilizes existing compression operations.

Another work discussed how to identify incompressible data to perform real-time compression [12]. To evaluate compressibility online, the authors used a method for compressing a portion of the data to measure compressibility and a method for conjecturing compressibility based on statistical values, such as byte entropy, core-set size, and serial correlation coefficients. Such statistical values are good indicators of compression efficiency. However, combinational compression does not consider the fact that the compression rate of one algorithm can affect the compression efficiency of another. Furthermore, these studies only discussed software-centric implementations. Logarithmic, square, division calculations, and subsequent decimal calculations demand significant computation time and hardware costs [55]–[57]. Our compressibility prediction method compensates for these points. The compression rate of Huffman encoding is evaluated by considering the effect of chunk size and LZ77 compression efficiency on Huffman encoding. Additionally, our compressibility measurement method is designed for standard hardware implementations. It reduces hardware costs by using several techniques. Our proposed method also uses statistical indicators that can be calculated with simple operations, such as shifting and absolute-value calculations.

A research conducted by Peter et al. proposed an adaptive compression system that searches for the most appropriate compression algorithm by considering the type of the target data and system conditions [13]. This system chooses the compression algorithm that maximizes network throughput based on CPU utilization and frequency, available network bandwidth, and data compressibility. Data compressibility is predicted by calculating the number of unique bytes that appear more than a certain threshold value in a data chunk. This approach differs from our approach in that it adaptively selects various compression algorithms considering system conditions, rather than selectively processing data using only a single compression algorithm considering compressibility.

IV. MOTIVATION

A. SYSTEM OVERVIEW

As shown in Fig. 4, the baseline DEFLATE compressor consists of an LZ77 encoder, Huffman code generator, and

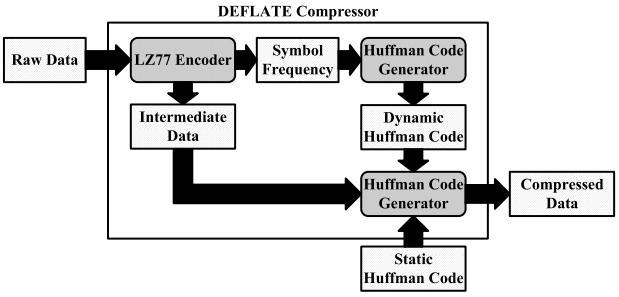


FIGURE 4. Block diagram of baseline DEFLATE compressor.

Huffman encoder. To handle data streams in parallel, data is segmented into DBs that can be processed independently.

The LZ77 encoder generates an LZ77 encoded stream (denoted as *intermediate data block (IDB)*) and calculates the frequency of the symbols (denoting as *symbol frequency*) appearing in a DB. The IDB and symbol frequency are transmitted to the Huffman encoder and Huffman code generator, respectively.

The Huffman code generator creates dynamic Huffman codes that are optimized for the DB currently being processed. After the dynamic Huffman codes are created, the optimal DEFLATE mode is selected by comparing the size of output data from each mode.

The Huffman encoder processes the original data or IDBs according to the selected DEFLATE mode. When a data chunk is compressed in the SBM, the Huffman encoder appends a header indicating the format of the corresponding data to the data chunk and arranges it according to the output memory configuration. When compressing in DEFLATE modes that include Huffman coding, the Huffman encoder compresses the IDB using the appropriate Huffman code.

B. MOTIVATIONAL DATA

To measure the compressibility of each DB accurately, both LZ77 encoding and dynamic Huffman code generation must be performed regardless of which DEFLATE mode is applied. However, SHM does not require Huffman code generation and SBM does not require both of Huffman code generation and LZ77 encoding process. Therefore, these processes inevitably waste computational resources.

Fig.5 illustrates the amount of unnecessary computations in the LZ77 encoder and Huffman code generator. For SHM and SBM, unnecessary computations represent the number of clock cycles consumed by the Huffman code generator or both of LZ77 encoder and Huffman code generator, respectively, to process DBs. Table 1 lists the types of benchmarks used for experiments. As shown in the experimental results, there are many unnecessary computations during DEFLATE compression, especially for incompressible benchmarks, such as *Media* and *Comp*. In the case of *Repeat*, most of the operations performed by the Huffman code generator are unnecessary because most DBs are compressed in SHM.

TABLE 1. Benchmarks.

Name	Description
Canterbury [59]	Canterbury Corpus. Standard benchmarks for data compression. Contains seven different types of files.
Calgary [59]	Calgary Corpus. Includes nine different types of files.
Text [60]	English novels. Contains only textual data. Expected to be compressed well.
Repeat [59]	Files containing significant repetition. Expected to be compressed well.
Bin [61]	Air/sea flux data. Data pattern is random but includes many repetitions. Expected to be compressed well by LZ77 encoding.
Media	Multimedia data. Already compressed or encrypted. Expected to be incompressible.
Comp [62]	Tropospheric data. Uses GRIB2 format. Expected to be incompressible because files are already compressed.

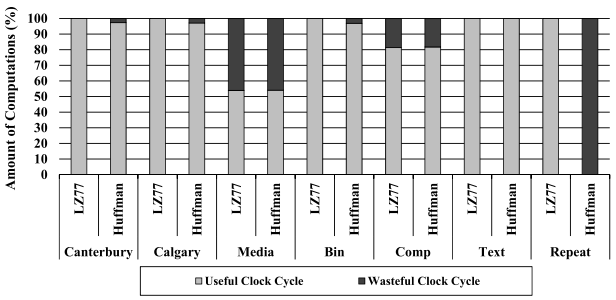


FIGURE 5. Wasteful clock cycle count.

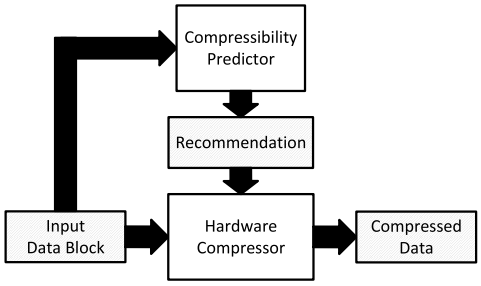
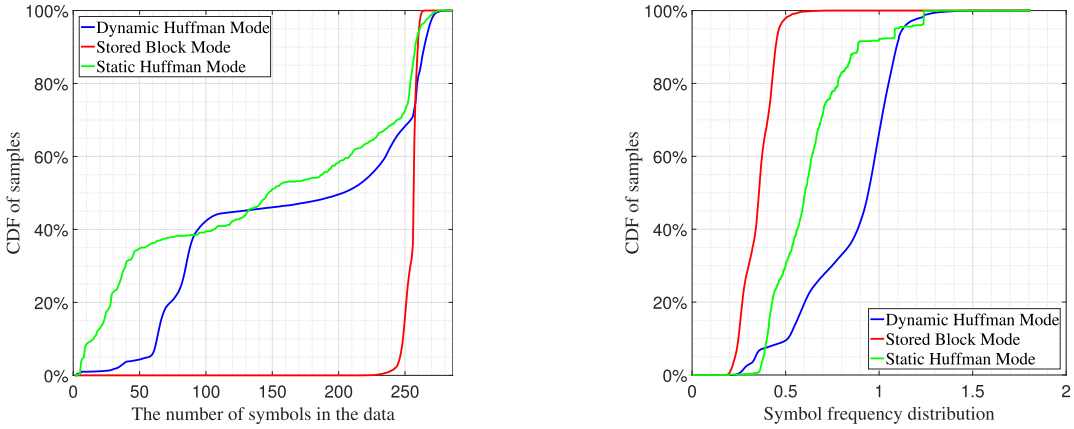


FIGURE 6. Simple overview of the selective compressor.

V. DESIGN EXPLORATION

A. DESIGN OVERVIEW

Fig.6 shows a system overview of the selective compressor. It receives DBs and processes data according to the compressibility of the DB. The system largely comprises two parts: compressibility predictor and compressor. The compressibility predictor predicts the compressibility of the input data chunk to determines whether to compress the data chunk. In detail, the compressibility predictor extracts features that reflect compressibility from the input DB. The predictor also



(a) symbol count CDFs for each DEFLATE mode

(b) symbol frequency CDFs for each DEFLATE mode

FIGURE 7. Symbol count CDFs for each DEFLATE mode.

analyzes the extracted features and generates information about whether the hardware compressor can compress the database. The hardware compressor receives the recommendation and processes the DB accordingly.

B. KEY TECHNIQUES

Our design goal is to enable the compressor to compress data with low computation and low hardware usage. To achieve the design goals, we propose two key techniques. First, we improve the accuracy of the algorithm of compressibility prediction to reduce the computations required to compress incompressible data. Next, we improve the resource performance.

1) ENHANCED HEURISTIC-BASED COMPRESSIBILITY PREDICTION

Compressibility prediction utilizes features that are relevant to the behavior of each algorithm, because each algorithm decides whether to compress the data. Most of the lossless compression algorithms are based on Huffman encoding or LZ77 encoding [12], [27]. Features related to LZ77 encoding include pair distance from random distribution [12] and standard deviations of the differences of consecutive bytes and XORed value of consecutive bytes [45], [46]. Features related to Huffman coding includes symbol count, symbol frequency distribution, etc. [12], [13], [45], [47].

Combinational compression generally consists of a dictionary-based compression and statistical compression [12], [27]. In the case of the DEFLATE algorithm, the input data is first compressed using LZ77 encoding, and it is then compressed using Huffman encoding. Previous studies have evaluated the compression efficiency of each compression algorithm. However, they did not consider the fact that the compression efficiency of the preceding compression algorithm could affect the compression efficiency of the subsequent compression algorithm.

The general DEFLATE algorithm performs Huffman encoding after the LZ77 encoder processes enough input data streams so that an intermediate data block of a certain size is created. However, chunk-based compression [48], [49] divides the data stream into a number of fixed-size chunks, which are compressed independently. In this case, the input data size of the Huffman encoder changes according to the compression efficiency of the LZ77 encoding. Thus, the compression efficiency of LZ77 encoding affects the compression efficiency of Huffman encoding.

Fig.7 presents the change of compression efficiency in the Huffman coding stage of chunk-based DEFLATE compression according to the number of symbols and the symbol frequency distribution. Generally, the smaller the number of symbols appearing in the DB and the higher the distribution between the frequencies of occurrence of each symbol, the higher the compression efficiency of Huffman encoding. However, as shown in figure, there are intervals that violate this tendency. This is because the input data size of Huffman encoding changes.

Fig. 8 shows the change of compression efficiency according to the input size of Huffman coding. As shown in figure, the compression efficiency decreases as the size of the input data block decreases. If the compression efficiency of Huffman coding is not good, there will be cases where the output data is larger than the input data, owing to compression. Sometimes Huffman coding causes the output data to be larger than the input data. In this case, Huffman coding does not save space; it just wastes computations.

Our compressibility prediction algorithm considers Huffman input size with existing features to determine whether Huffman coding should be performed for the input DB. DEFLATE uses SBM, which does not generate Huffman codes when the size of output data is bigger than original Huffman coding. Therefore, our proposed compressor, using the DEFLATE algorithm, selects SBM by considering compressibility prediction algorithm as well as existing features and Huffman input size.

Compressibility prediction usually precedes data compression. However, the size of the intermediate data block can be

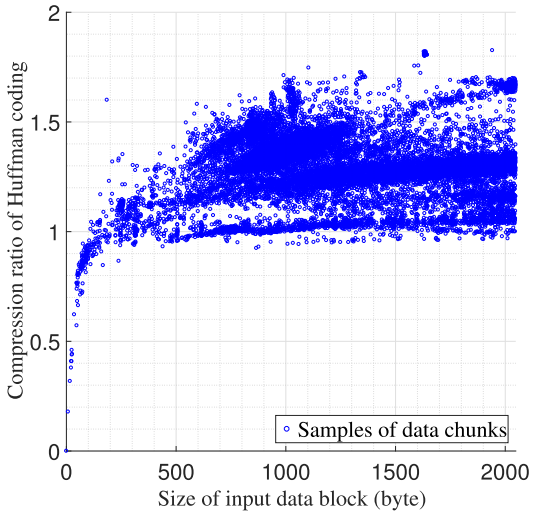


FIGURE 8. Compression efficiency by input size of Huffman coding.

known after LZ77 encoding is finished. Therefore, it may be difficult to reflect the size of intermediate data input to the Huffman encoder in the compression-rate prediction. In this case, the size of the input data block to the DEFLATE compressor and the predicted LZ77 encoder compression efficiency can be used instead. In the proposed compressor, owing to the hardware cost-reduction technique, the compressor predicts compressibility after LZ77 encoding. Therefore, our compressor uses the actual size of the intermediate data block. The implementation details are given in the implementation issue section.

2) HARDWARE COST-REDUCTION TECHNIQUE

Implementing a compressibility predictor requires hardware resources. The purpose of compression is to reduce wasted computation and ultimately improve system throughput by selectively bypassing incompressible DBs. However, because hardware resources are limited, using large amounts of hardware resources for compressibility prediction can also hinder throughput improvements. As can be seen in Fig. 6, applying selective compression to the actual DEFLATE compressor results in a 14.1% increase in hardware resource consumption. There is little benefit from selective compression, especially when compressible data is entered consecutively. In this case, the performance of a compressor without a compressible predictor might be better. Thus, hardware resource overhead required to apply compressibility prediction should be minimized.

In this paper, two methods are used to achieve low resource usage.

- **Computation reuse for feature extraction:** first, when applying selective compression, we reuse the computations of the existing compression algorithm. The features used to determine compressibility are mainly probability statistical indicators, such as the number of symbols in the data, the frequency of occurrence of the symbols, and the variance of the frequencies. Statistical compression also works based on the probability of occurrence of each occurrence symbol. Generally, dictionary compression mostly precedes statistical compression when using combinational compression algorithms [27]. Therefore, generation of symbol occurrence probability must be preceded. In other words, most of the operations that scan data and calculate the number and frequency of symbol occurrences used to calculate these features include dictionary compression. By using this, the resource usage of the compressibility predictor can be reduced considerably.
  The DEFLATE algorithm collects statistics during LZ7 encoding and sends them to the Huffman coding phase. Taking advantage of this, we implement the feature-extraction feature utilizing the statistics-collection process performed in parallel with LZ77 encoding. In addition, symbol frequency distribution calculation is implemented using some Huffman encoder operations. The implementation details are given in the implementation issues section.
- **Computation complexity reduction:** many previous researches have studied compressibility prediction [12], [13], [22], [45]–[47]. However, these studies only discussed software-centric implementations, such as log, square, division, and floating-point operations, all requiring significant computational time and hardware costs. In this paper, we reduce the computational complexity of compressibility prediction to minimize this overhead. The detailed implementation description is also given in the implementation issues section.

VI. IMPLEMENTATION DETAILS

This section describes our selective data compression system based on data compressibility prediction. We apply the compressibility prediction to the DEFLATE algorithm. Specifically, the compressibility prediction enables the early selection of the DEFLATE mode. The proposed compressor design reflects the key techniques introduced earlier.

First, we explain the data flow of the proposed system and major design goals. We then analyze the factors affecting DEFLATE compression efficiency. Finally, we discuss our compressibility prediction technique.

A. OVERVIEW OF PROPOSED SYSTEM

Fig. 9 illustrates the data flow in the proposed scheme. This scheme is largely divided into two phases: the training phase and compression phase. Features are statistical indicators that represent data compressibility. The training phase extracts features from the training data for predicting data compressibility and builds a predictive model by analyzing the extracted feature sets. The prediction model learns to select the optimal DEFLATE mode based on the variation of each feature. The training phase may be performed only once prior to the compression phase. The compression phase includes *feature-extraction* and

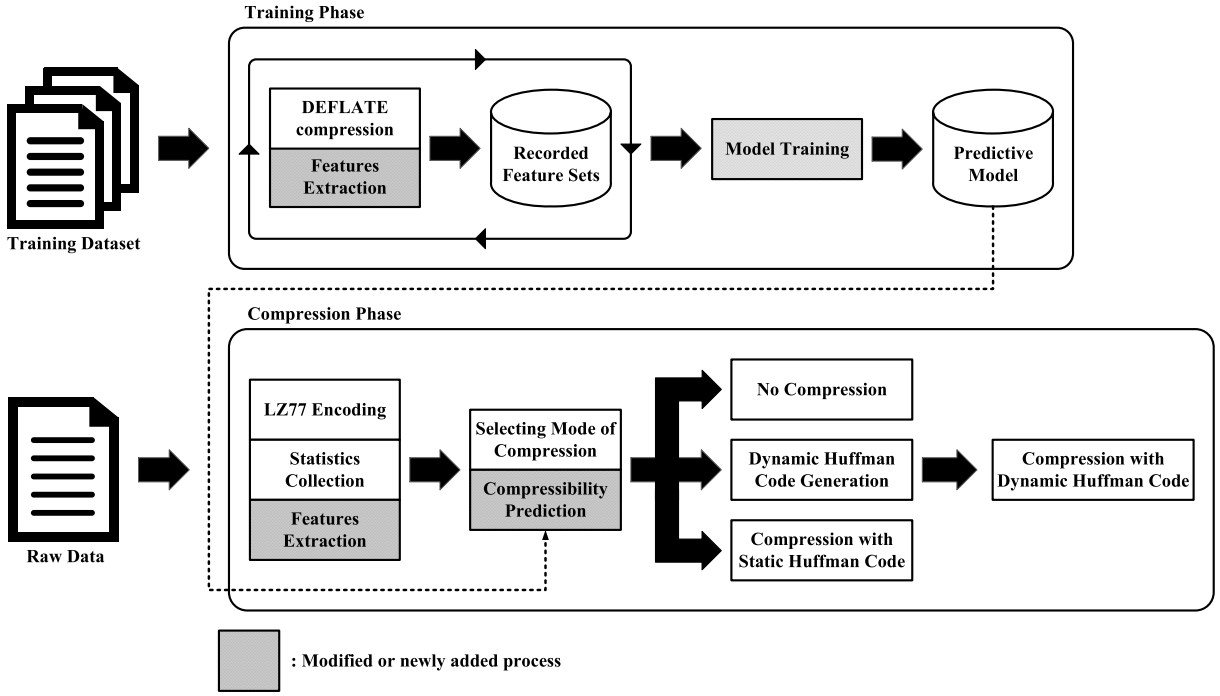


FIGURE 9. Block diagram of proposed DEFLATE compression system.

compressibility prediction. Feature-extraction calculates the statistical indicators for predicting the compressibility of current DBs. Compressibility prediction predicts the compressibility of the DBs by using the predictive model that was generated in the training phase. Finally, the compressor selects the appropriate DEFLATE mode according to the prediction result and performs the remaining compression operations.

B. ANALYSIS OF DEFLATE COMPRESSION RATE

This section analyzes the factors that affect the DEFLATE compression ratio in terms of accurate data compressibility prediction. Table 2 lists the necessary terminology. The data compression rate of each DEFLATE mode is defined as follows.

- $CR_{dyn\_mode} \simeq CR_{LZ77} \times CR_{dyn\_Huff}$
- $CR_{stt\_mode} \simeq CR_{LZ77} \times CR_{stt\_Huff}$
- $CR_{str\_mode} \simeq 1$

Because the size of the header that contains information regarding DBs is negligible, it is not considered. First, we calculate the data compression ratio when using DHM. The LZ77-encoded IDB consists of literals and length-distance pairs. Therefore, the size of the IDB is defined as follows.

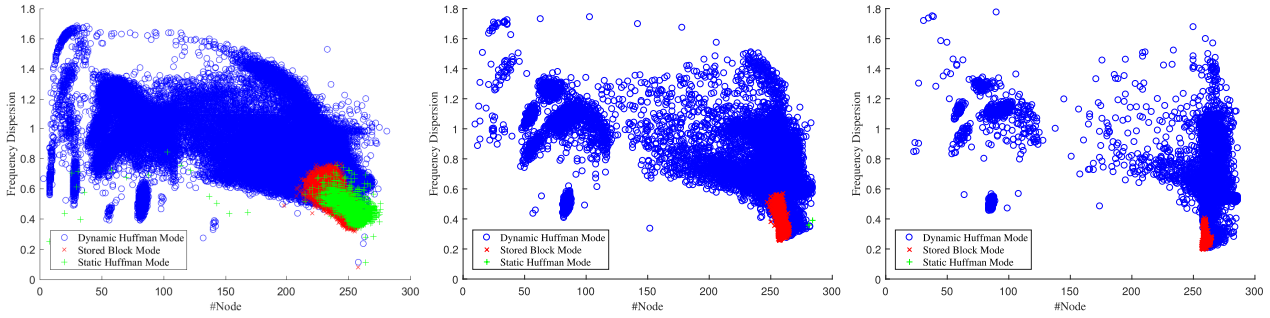
$$(bit_L \times L) + (bit_P \times P)$$

Length-distance pairs are encoded using literal-length Huffman codes, distance Huffman codes, and the extra bit required to extrapolate certain Huffman codes to their original values. The average length of the dynamic Huffman codes is close to the entropy of the symbols [52]. Therefore, when an IDB is encoded, the size of the data is defined as follows.

$$tree + (ent_L \times L) + [(ent_L + ext_L) + (ent_D + ext_D)] \times P$$

TABLE 2. Major parameters for analysis of the DEFLATE compression rate.

Name	Representation
$tree$	size of dynamic Huffman tree
$CR_{(dyn/stt/str)\_mode}$	compression rate of (DHM/SHM/SBM)
$CR_{(LZ77/Huff)}$	compression rate of (LZ77/Huffman) encoding
$CR_{(dyn/stt)\_Huff}$	compression rate of (dynamic/static) Huffman coding
L/P	the number of (literal/length-distance pairs)
$bit_{(L/P)}$	bit length of (literal/length-distance pairs)
$ent_{(L/D)}$	average entropy of (literal-length/distance)
$ext_{(L/D)}$	average extra bit length of (literal-length/distance)
$Frequency_i$	frequency of i-th symbol
$Frequency_{(avr/dis)}$	(average/dispersion) of frequencies
$\#Node$	the number of Huffman nodes
$Data_{(db/idb)}$	size of (DB/IDB)
$Data_{cri\_db}$	critical IDB size (CIDBS)
CR_{cri_LZ77}	critical LZ77 compression rate (CLZCR)
L_code_{static}	average length of static literal-length Huffman codes



(a) $0KB < Data_{idb} \leq 1KB$ (b) $1KB < Data_{idb} \leq 2KB$ (c) $2KB < Data_{idb} \leq 4KB$

FIGURE 10. $0KB < Data_{idb} \leq 1KB$.

Therefore, the data compression ratio of Huffman coding is defined as follows.

$$\frac{(bit_L \times L) + (bit_P \times P)}{tree + ent_L \times L + (ent_L + ext_L + ent_D + ext_D) \times P}$$

Because most compressible DBs use DHM, there is no need to measure DEFLATE compression efficiency accurately when DEFLATE compression efficiency is high. However, the compression efficiency of incompressible data must be accurately measured. For this reason, we assume that the compression efficiency of LZ77 encoding is very low when the IDB contains very few length-distance pairs. If the number of length-distance pairs is zero, the data compression ratio of Huffman coding is defined as follows.

$$\frac{(bit_L \times L)}{tree + (ent_L \times L)} (\because P \approx 0)$$

Therefore, the compression rate of DHM is defined as follows.

$$\therefore CR_{dyn\_mode} = CR_{LZ77} \times \frac{bit_L}{\frac{tree}{L} + ent_L} \tag{2}$$

Based on the same principle, the compression rate of SHM is defined as follows.

$$\therefore CR_{stt\_mode} = CR_{LZ77} \times \frac{bit_L}{L\_code_{static}} \tag{3}$$

C. FEATURES

This section discusses the features used for predicting the compressibility of a DB. In our experiment, we used zlib [28] to compress various types of files. According to the DEFLATE compression ratio analysis presented above, the DEFLATE compression ratio is largely influenced by three main factors.

1) ENTROPY

According to (2), if ent_L increases, the DEFLATE compression rate decreases. This is because the larger ent_L is, the longer the average length of the literal-length Huffman codes, which reduces the compression effect on the data. Because significant hardware resources are required to implement floating point operations and logarithms [55], [56], we do not use entropy for compressibility prediction.

Instead, we focus on the two statistical indicators related to entropy. cardinality and dispersion [62]. Generally, the larger the cardinality of a set, the higher the entropy value of the set. Entropy is maximized when the probability distribution of a dataset is uniform. From the viewpoint of Huffman coding, the cardinality and dispersion of a dataset are the number of leaf nodes in the corresponding dynamic Huffman tree and degree of distribution between the frequencies of occurrence of each symbol (denoted as *frequency dispersion*), respectively. Both indicators are related to compression efficiency. The average depth of each leaf node increases as the number of leaf nodes increases, which results in a decrease in compression efficiency. Additionally, it is easy to generate efficient Huffman codes if some symbols appear more frequently.

Fig. 10a presents the distribution of DEFLATE modes selected by each DB according to their frequency dispersion and number of Huffman nodes. The experimental results reveal that the DEFLATE compression ratio decreases as the number of Huffman nodes increases and the frequency dispersion decreases. Based on these observations, the prediction algorithm considers the number of Huffman nodes and frequency dispersion of the DB as factors affecting compressibility measurements. This method is denoted as *entropy filtering*. Additionally, let *feature space (FS)* denote the space representing the DEFLATE mode distribution according to these two features. In the FS, the areas where the samples of DHM, SHM, and SBM are concentrated are searched and referred to as the *dynamic Huffman area*, *static Huffman area*, and *stored block area*, respectively.

2) SIZE OF DYNAMIC HUFFMAN TREE

When a data stream is encoded using dynamic Huffman codes, dynamic Huffman tree information must be appended to the output data. $\frac{tree}{L}$ is the term representing the increase in data size caused by the addition of dynamic tree information. According to (2), the effect of dynamic Huffman code information size on DEFLATE compression efficiency

becomes smaller as the number of literals (i.e., size of the IDB) increases. Because the size of a dynamic Huffman tree is difficult to predict, this paper only focuses on the effect of the size of the IDB on the DEFLATE compression ratio, rather than the dynamic Huffman tree.

As mentioned, the compressibility prediction algorithm for the existing DEFLATE algorithm does not consider the size of the IDB. However, using the chunk-based compression method, the compressibility prediction algorithm must consider it, because the IDB size varies drastically, depending on the compression efficiency of the LZ77 encoding. Fig. 10 presents the change in FS as the size of the IDB changes. As shown in Fig. 10, the smaller the IDB, the larger the stored block and static Huffman area. This is because as the IDB size increases, the overhead incurred by the addition of the dynamic Huffman tree information becomes larger, which results in degradation of the compression efficiency of dynamic Huffman coding. This indicates that as the size of IDB changes, and the condition determined to be incompressible DB changes. In other words, symbol-count and frequency-dispersion thresholds should be adjusted according to IDB size. Therefore, it is necessary to use different FSs according to the size of the IDB to predict data compressibility.

When the size of the IDB is very small, the probability of using static Huffman coding instead of dynamic Huffman coding increases, even if the compression efficiency of the dynamic Huffman code is very high. This is because the dynamic Huffman tree size is relatively large compared to the compressed data.

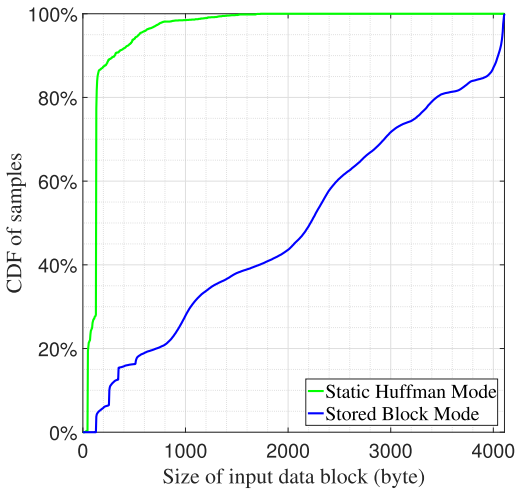


FIGURE 11. CDF graph of DHMDB and SHMDB numbers according to IDB size.

When compressed in DHM and SHM, the best compressed data blocks will be marked as *DHMDB* and *SHMDB*, respectively. Fig. 11 shows the CDF of the number of DHMDB and SHMDB, depending on the size of the IDB based on the dataset described in Table 1 and some other 1.9 GB random datasets belonging to the same categories. As shown in the figure, almost all SHMDBs have very small IDBs. This information is utilized to formulate the threshold with which the compressibility prediction algorithm preferentially selects the SHM. Thresholds are set heuristically in the same way as in previous research [12]. Specifically, the point where 80% of the data blocks for which the selected SHM belongs is set as IDB threshold. Below this threshold, 80% of DBs that choose SHM and 4% of DBs that choose DHM also belong.

3) COMPRESSION RATE OF LZ77 ENCODING

Due to the size of the dynamic Huffman tree information, the size of the output data may be larger than the size of the original data. In this case, the DEFLATE algorithm handles the DB in SBM. However, the increase in data size caused by Huffman coding is limited. Therefore, if the compression rate of LZ77 encoding is higher than a certain threshold, the DB is not compressed in SBM. This condition can be expressed as follows.

$$CR_{LZ77} \times CR_{Huff} > 1 \tag{4}$$

$$\iff CR_{LZ77} > \frac{1}{CR_{Huff}} \tag{5}$$

Fig. 12 illustrates the fact that the DEFLATE algorithm does not use SBM when the compression rate of LZ77 is above the threshold. Based on this observation, the proposed algorithm excludes SBM from mode selection when the LZ77 encoding compression rate of the DB is more than the threshold. During the training phase, the maximum value of $\frac{1}{CR_{Huff}}$ is recorded. This value is called the *CLZCR*.

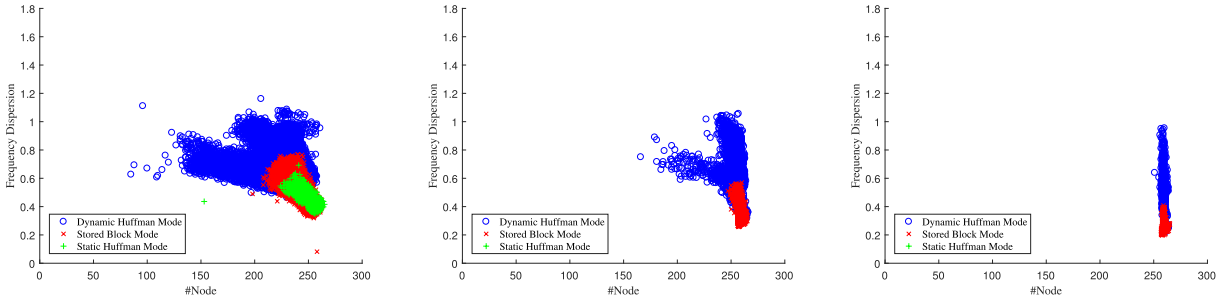
D. PROPOSED METHOD

This section describes the proposed data compressibility prediction scheme. Specifically, we describe two phases included in the proposed method: the training phase and compression phase.

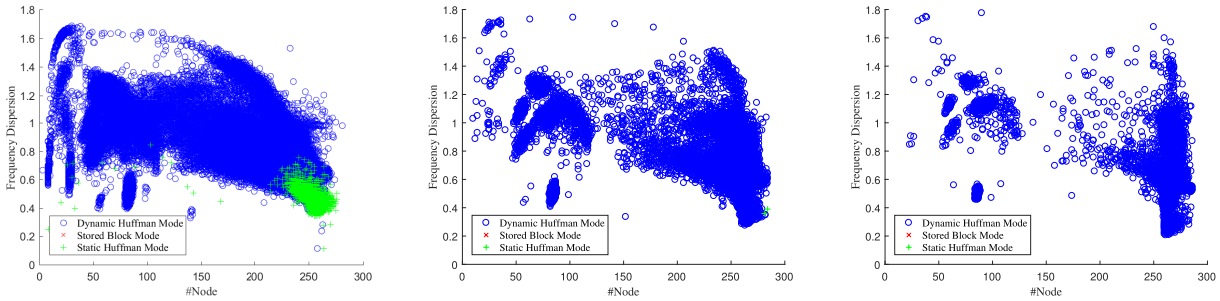
1) TRAINING

As discussed above, the training phase extracts features from the training data and builds a predictive model. The training data should consist of a sufficiently large dataset of various data types to ensure the universality of the data. Fig. 14 presents a block diagram of the training phase. While training data is processed, a feature set for each DB is generated. All extracted feature sets are recorded. The feature sets include the following statistics: $Data_{idb}$, CR_{Huff}, $\#Node$, $Frequency_{dis}$, DEFLATE mode, *tree*, and ent_L.

The recorded feature sets are classified into different feature groups according to predetermined IDB size ranges and the CLZCR. Classification is performed twice. Each feature group has one FS and a CIDBS. The first classification is performed based on the intermediate data size as soon as the feature set is extracted. After feature-extraction iteration is complete, the recorded feature sets belonging to each intermediate feature group are reclassified into feature groups based on their CLZCRs. The CLZCR is calculated as



(a) $0KB < Data_{idb} \leq 1KB$ and $CR_{LZ77} \leq CR_{cri\_LZ77}$ (b) $1KB < Data_{idb} \leq 2KB$ and $CR_{LZ77} \leq CR_{cri\_LZ77}$ (c) $2KB < Data_{idb} \leq 4KB$ and $CR_{LZ77} \leq CR_{cri\_LZ77}$



(d) $0KB < Data_{idb} \leq 1KB$ and $CR_{LZ77} > CR_{cri\_LZ77}$ (e) $1KB < Data_{idb} \leq 2KB$ and $CR_{LZ77} > CR_{cri\_LZ77}$ (f) $2KB < Data_{idb} \leq 4KB$ and $CR_{LZ77} > CR_{cri\_LZ77}$

FIGURE 12. $0KB < Data_{idb} \leq 1KB$ and $CR_{LZ77} \leq CR_{cri\_LZ77}$.

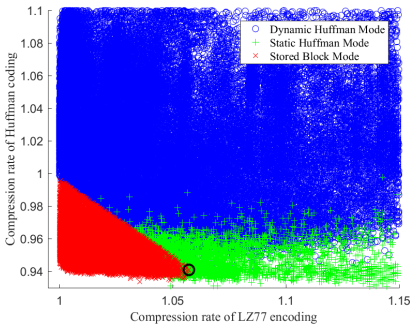


FIGURE 13. Distribution of DEFLATE modes by compression rate of LZ77 encoding and compression rate of Huffman coding.

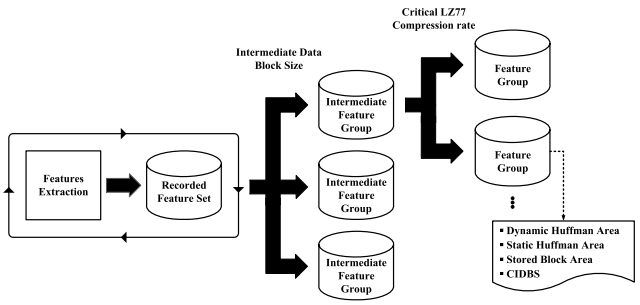


FIGURE 14. Block diagram of the training phase.

the maximum value of $\frac{1}{CR_{Huff}}$ in each intermediate feature group. Following the second classification, the CIDBS of each feature group is calculated using the features belonging to each feature set. The CIDBSs are computed as the point where the CDF of the DB sample is 80% according to the IDB size. Additionally, the DEFLATE areas for each feature group are defined.

In this paper, we define the three areas as rectangles for simplicity of algorithm. Each rectangle contains data clusters in which the data samples for each DEFLATE mode are continuously distributed in the FS. The rectangles are defined as the regions between the maximum and minimum values of the number of Huffman nodes of each data cluster and are smaller than the maximum value of the frequency dispersion of each data cluster. The dynamic Huffman area is the remaining area excluding the other two areas.

2) FEATURE EXTRACTION

The DEFLATE compression system extracts the features that are necessary for compressibility prediction during LZ77 encoding. The features extracted through this process are $Data_{idb}$, $\#Node$, $Frequency_{dis}$, and CR_{LZ77}.

3) COMPRESSIBILITY PREDICTION

Alg. 1 is the compressibility prediction algorithm. The compressibility prediction algorithm selects the appropriate FS depending on the size of each IDB and the LZ77 compression ratio. Next, entropy filtering is performed using the selected FS. First, the algorithm determines which area the current DB belongs to base on the number of Huffman nodes and frequency dispersion. It is assumed that all DBs

Algorithm 1 Entropy Filtering

Input: Attributes of current DB
Output: DEFLATE mode

```
FS ← FS_Select(Data_idb, CR_LZ77);
Area ← Area_Select(FS, #Node, Frequency_dis);
if Area == StoredBlock then
    return SBM
else
    if Data_idb < FS.Data_cri_db then
        return SHM
    else
        if Area == Static then
            return SHM
        else
            return DHM
        end if
    end if
end if
```

belonging to a specific area are compressed in the corresponding DEFLATE mode. However, in the FS, multiple areas overlap. In this case, the mode with the highest priority is selected. The selection priority decreases in the order of SBM, SHM, and DHM. This is because the computational cost decreases in the order of SBM, SHM, and DHM. Therefore, if the current DB belongs to the stored block area, the DB is processed in SBM. Otherwise, the algorithm calculates whether or not the IDB size of the DB is less than the CIDBS. If the IDB size is smaller than the CIDBS, the current DB is compressed in SHM. If not, the DEFLATE mode is selected according to the area to which the current DB belongs.

VII. IMPLEMENTATION ISSUES

In this section, we discuss the hardware implementation issues related to the proposed method. Specifically, we describe how the proposed hardware cost-reduction techniques are implemented. We also use a pipeline technique to optimize performance when implementing the proposed method.

A. COMPUTATION REUSE FOR FEATURE EXTRACTION

As mentioned above, we found similarities between the computation of compressibility prediction and the computation of existing compression algorithms and used them to minimize resource consumption. Reusing computation is a simple but very efficient method to reduce resource consumption. This technique can also be applied to combinatorial compression, which includes statistical compression that generates code based on static tables that record the frequency of symbols occurrences. This subsection describes the operations of compressibility prediction can be eliminated.

The features extracted via the feature extraction process are as follows: $Data_{idb}$, $\#Node$, $Frequency_{dis}$, and CR_{LZ77}.

1) SIZE OF INTERMEDIATE DATA BLOCK ($Data_{idb}$)

The Huffman encoder needs to know how much intermediate data should be compressed with the Huffman code table that is currently transferred. Therefore, $Data_{idb}$ is calculated in the statistics-collection process, and the information is sent to the Huffman encoder.

2) NUMBER OF HUFFMAN NODES($\#Node$)

Because the Huffman code generator needs to know how many nodes to generate dynamic Huffman code, $\#Node$ is also generated during the existing statistics collection.

3) COMPRESSION RATE OF LZ77 ENCODING (CR_{LZ77})

CR_{LZ77} is the input block size ($Data_{db}$) of the LZ77 encoding divided by the size of the currently generated intermediate data block ($Data_{db}$). As mentioned above, $Data_{idb}$ is created during the statistics collection process, and only the value of $Data_{db}$ needs to be generated. Fig. 15 illustrates the layout of the compressed data when using each DEFLATE mode [40]. For SBM, the length of the original data block is included in the compressed data block. Therefore, in the case of the compressor supporting SBM, $Data_{db}$ is calculated to be transferred to the Huffman encoder during the statistics collection process. It is assumed that our compressor supports SBM.

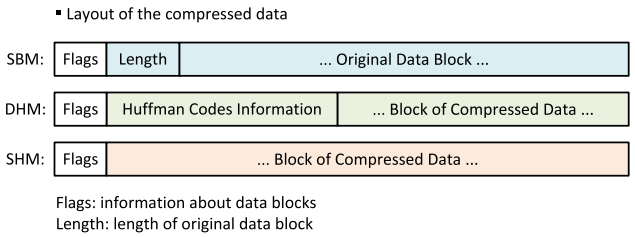


FIGURE 15. Layout of the compressed data.

4) DISPERSION OF FREQUENCIES ($Frequency_{dis}$)

Finally, a calculation of $Frequency_{dis}$ is required. $Frequency_{dis}$ can be calculated using the following equation:

$$\frac{\sum_{i=1}^{\#Node}(Frequency_i - Average\_Frequency)^2}{\sum_{i=1}^{\#Node} Frequency_i} \tag{6}$$

To calculate $Frequency_{dis}$, the average frequency must be calculated. Average frequency is the summation of symbol frequencies divided by the number of occurred symbols ($\#Node$). Average frequency can be calculated using following equation:

$$\frac{\sum_{i=1}^{\#Node} Frequency_i}{\#Node} \tag{7}$$

The statistics collection process includes scanning the input data stream, calculating the frequency ($Frequency_i$) of each occurred symbol, and calculating the number of occurred symbols ($\#Node$). Additionally, Huffman encoding fetches the frequencies of all symbols to sort the Huffman

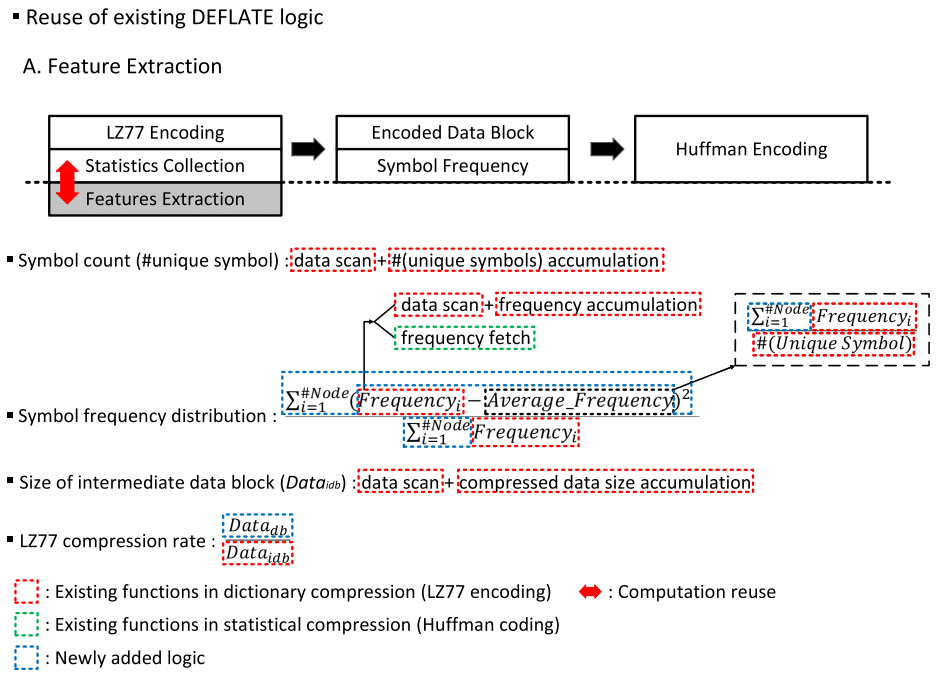


FIGURE 16. Operations removed by the computation-reuse technique.

nodes by frequency. Therefore, apart from these calculations, the following calculations only require additional logic.

- Accumulating all symbol frequencies
- Accumulating the square of the difference between the frequency of each symbol and average frequency
- Division operations

Among these, computation complexity-reduction simplifies second operations and replaces third operations with simple shift operations.

Fig. 16 describes the results of applying the computation-reuse technique. The red and green parts contain the calculations implemented by reusing the existing DEFLATE operations. Therefore, for compressibility prediction, only the blue parts needs to be additionally implemented.

B. PIPELINING FREQUENCY DISPERSION CALCULATION

The frequency dispersion calculation can be initiated after feature extraction is complete. To calculate the frequency dispersion, the frequencies of all symbols must be read and the cumulative calculation results of the numerator and denominator in (6) must be obtained.

Therefore, the frequency dispersion calculation takes a relatively longer time compared to the other feature calculation steps, which may degrade compression performance. To prevent performance degradation, we parallelized frequency dispersion calculation and Huffman code generation.

To generate dynamic Huffman code, the Huffman code generator searches for the two Huffman nodes having the smallest frequency values. The DEFLATE algorithm sorts the Huffman nodes according to their frequencies to search for these two Huffman nodes [28]. The frequencies of all symbols are read for the sort operation. Simultaneously, the frequency dispersion is calculated in parallel while sorting the Huffman nodes. After the frequency dispersion calculation is finished, the DEFLATE mode is selected. If DHM is selected, the rest of the compression process is performed. In other cases, Huffman-code generation is aborted. The Huffman encoder then processes the current DB in the selected DEFLATE mode. Fig. 17 presents an example of a pipelining frequency dispersion calculation.

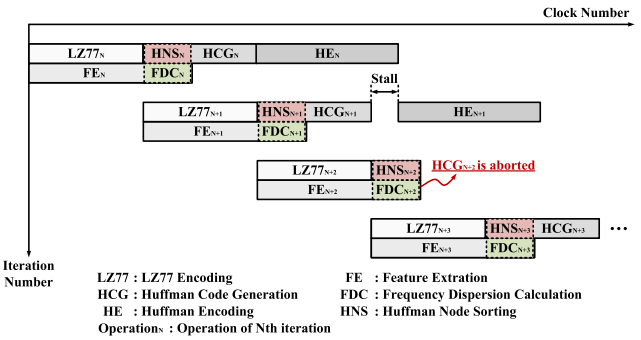


FIGURE 17. Space-time diagram of the proposed compressor pipeline.

C. COMPUTATION COMPLEXITY REDUCTION

1) FREQUENCY DISPERSION CALCULATION

Statistical variation is a measure of how datasets are distributed. There are various types of indicators for statistical dispersion. A general statistical indicator of how far

each item in a set is separated from the mean is variance. When frequency dispersion is defined as the variance of each symbol frequency, it is calculated as follows:

$$\frac{\sum_{i=1}^{\#Node}(Frequency_i - Average\_Frequency)^2}{\sum_{i=1}^{\#Node} Frequency_i} \quad (8)$$

In this case, multipliers and dividers that consume significant hardware resources are necessary to perform the square operation. However, average absolute deviation can be calculated using the following equation:

$$\frac{\sum_{i=1}^{\#Node} |Frequency_i - Average\_Frequency|}{\sum_{i=1}^{\#Node} Frequency_i} \quad (9)$$

Average absolute deviation uses an absolute operation that requires two subtractions and multiplexing operations. Therefore, using absolute average deviation instead of variance as an indicator of frequency dispersion reduces hardware resource utilization.

D. COMPRESSIBILITY PREDICTION

In Alg. 1, FS_Select and Area_Select perform the following operations:

$$\frac{Data_{db}}{Data_{idb}} \geq \delta \quad (10)$$

$$\frac{\sum_{i=1}^{\#Node} |Frequency_i - Average\_Frequency|}{\sum_{i=1}^{\#Node} Frequency_i} \leq \alpha \quad (11)$$

As shown above, division, which consumes significant hardware resources [57], is necessary for (10) and (11). Furthermore, considering the operand types, floating-point operations are required, which further increases hardware resource utilization [55]. To solve this problem, we convert the division operations into multiplication operations and approximate all real operands as the sum of powers of $\frac{1}{2}$. Experiments have shown that this approach can reduce LUT usage by approximately 82.9% compared with using dividers. Because multiplication of $\frac{1}{2}$ can be converted into an arithmetic right-shift operation, all real divisions are converted into the sum of integer multiplications and right-shift operations. If a real number value is approximated using this method, an error may occur in the calculation result. However, when approximating a decimal using the sum of $(\frac{1}{2})^k$ such that $1 \leq k \leq n$, the error, e, is $0 \leq e < (\frac{1}{2})^n$, which is negligible. A proof of this operation has been omitted owing to lack of space.

VIII. EXPERIMENTAL RESULTS

A. EXPERIMENTAL ENVIRONMENT

To evaluate the proposed technique, we implemented a DEFLATE compressor with the proposed compressibility prediction technique in Verilog register-transfer-level code. We synthesized the compression engine using Xilinx Vivado. The target FPGA was the xcku025-ffva1156-1-I. The synthesized design operates at a target frequency of 200 MHz. Training for the compressibility prediction technique was performed using a modified version of zlib [28].

The baseline system for this experiment consisted of the LZ77 encoder, Huffman code generator, and the Huffman encoder. The three modules operate in a pipelined manner. In this experiment, we measured the performance changes in the Huffman code generator and Huffman encoder, which are affected by the compressibility prediction technique. Additionally, we discuss the effects of performance changes in the two modules on overall compressor performance. Therefore, the LZ77 encoder should not be a performance bottleneck and the proposed compression system implements two LZ77 encoders to support one Huffman code generator and one Huffman encoder. Also, the data to be compressed is divided into 4-KB chunks so the two LZ77 encoders can operate in parallel.

B. BENCHMARKS

Table 1 lists the types of files used for experimentation. During the training phase and data compression process, different files belonging to the same category, except for the Canterbury and Calgary files, were used. The files used for training and data compression were 320 MB and 32 MB, respectively.

Fig. 18 presents the ratio of DEFLATE modes selected by the DBs included in each benchmark when the benchmarks were compressed. Based on the results of this experiment, the benchmarks were divided into five categories: *normal*, *txt*, *repeat*, *bin*, and *incomp*). The normal category includes the Canterbury and Calgary files.

Normal contains various types of files ranging from text files to executable files. Most of the files are compressed in DHM and are relatively compressible compared to the other benchmarks.

Txt files are compressed well because they are composed of a limited set of characters, such as alphanumeric characters. Huffman compression efficiency is relatively high for these files. All DBs in *txt* select DHM.

Repeat files have data patterns in which several characters are repeated many times. For this reason, these files have very high LZ77 encoding compression efficiency. They are also compressed well by dynamic Huffman coding because they contain only alphabet characters. However, based on the high compression efficiency of the LZ77, most files are compressed in SHM, which does not require appending dynamic Huffman trees to the output data stream.

In the case of *bin*, LZ77 compression efficiency is high because it contains significant repeated data. However, the compression efficiency of Huffman coding is not good because of the variety of characters appearing in the files. Therefore, when the files are compressed, SHM is selected relatively more often compared to the other benchmarks.

Incomp includes multimedia files and weather information data. Because multimedia files and weather data are already compressed, they are incompressible. Experimental results revealed that *incomp* files contain many DBs that select SBM.

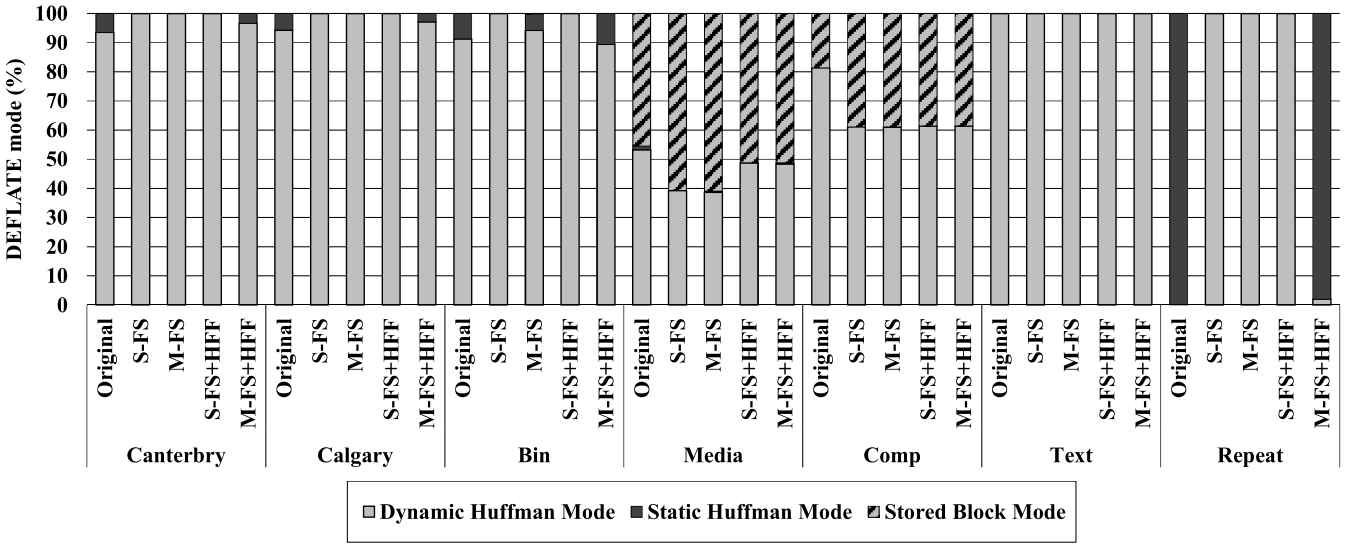


FIGURE 18. DEFLATE modes percentages selected by the DBs included in each benchmark.

C. TYPES OF ENTROPY FILTERING

To investigate the effects of features on the proposed compressibility prediction method, we performed entropy filtering in several different ways. We used four different entropy filtering schemes. The first scheme measures compressibility based on the frequency dispersion and number of Huffman nodes, and is called *single FS (S-FS)*. The next scheme uses multiple FSs based on the sizes of the IDBs and is called *multiple FS (M-FS)*. *feature filtering (FF)* measures compressibility using the CIDBS and LZ77 encoding compression ratio in addition to *S-FS*. Finally, the scheme using all variables is called *M-FS+FF*.

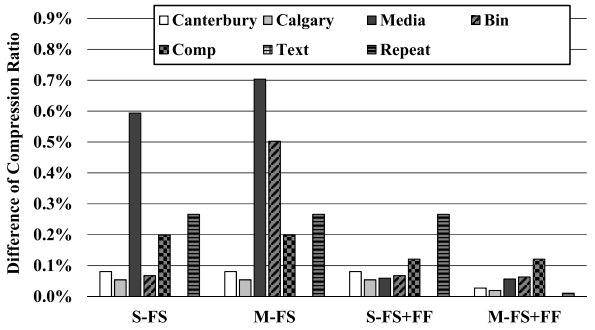


FIGURE 19. Compression rate differences compared to original method based on the inclusion of entropy filtering schemes.

D. ACCURACY

Fig. 19 presents the changes in DEFLATE mode and DEFLATE compression ratio when the benchmarks were compressed with various entropy filtering schemes. We derived several observations from this experiment.

(a) Two experimental results show that the prediction accuracy of *M-FS+FF* is the best. Additionally, it is shown that there is no significant difference in the DEFLATE compression ratio when the compressibility prediction method is used.

(b) *M-FS* considers more features than *S-FS*, but often has lower accuracy than *S-FS*. This can be observed for the *media*, *comp*, and *bin* categories. This is because the overlap between multiple areas becomes large when the IDB size is small for *M-FS*. This increases the number of DBs that select the high-priority DEFALTE mode. As shown in the figure, when *M-FS* compresses *bin* and *media*, the ratio of the DBs that select SBM increases compared to that in *S-FS*. In the case of *comp*, the percentage of DBs selecting SHM increases.

(c) The compressibility prediction accuracy of *S-FS+FF* and *M-FS+FF* is better than that of *S-FS* and *M-FS*, respectively. *S-FS+FF* and *M-FS+FF* predict compressibility more accurately in overlapping areas by considering the CIDBS and CLZCR. For this reason, especially in benchmarks that contain many DBs belonging to overlapping regions, such as *incomp*, prediction accuracy is greatly improved by FF.

(d) The *normal* and *repeat* categories show no change in prediction accuracy with entropy filtering, except for *M-FS+FF*. This is because all features must be used for compressibility prediction to select a static Huffman area in overlapping regions.

We evaluate the effects of reflecting the size of IDB in compressibility prediction. To evaluate this, benchmarks containing SHMDBs are collected and compared before and after compressibility predictions considering IDB size. The frequency dispersion and the number of symbols that are used as features for predicting data compressibility in S-FS are the features widely used in previous studies [12], [45]–[47]. Therefore, set S-FS to baseline. SHMDB is the DB that can be best compressed when compressed with SHM. Fig. 20 shows the change in SHMDB that is selected when multiple FSs are used for S-FS and CIDBS is used for filtering. As you

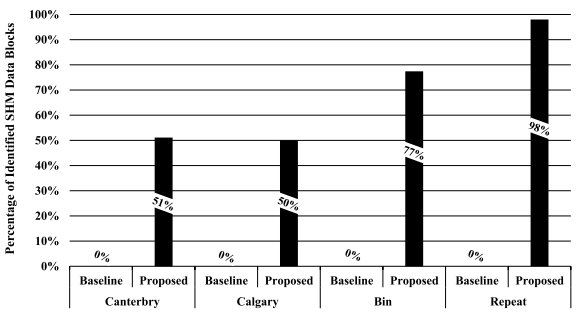


FIGURE 20. Accuracy of the SHMDB detection rate.

can see in Fig. 20, when using baseline, there is no DB to select SHM. On the other hand, considering the size of IDB, we can see that 71.3% of SHMDB is processed in SHM. This means that the proposed compressibility prediction algorithm evaluates the compression efficiency of Huffman encoding more accurately than the existing algorithm.

E. PERFORMANCE

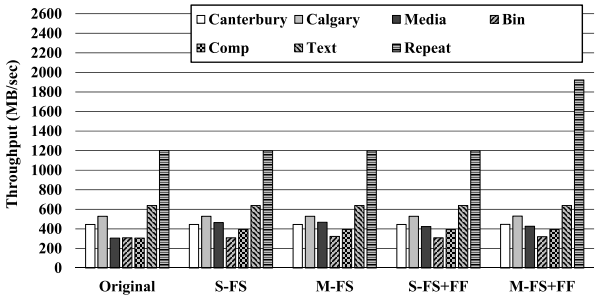
Fig. 21a illustrates the throughput of the compressor for each entropy filtering scheme. Additionally, Figs. 21b and 21c present the average clock cycles consumed by the Huffman code generator for processing a DB and the clock cycles for which each module was stalled by processing delays in the other modules.

In general, the proposed technique improves compressor throughput. In the cases of *incomp* and *repeat*, the proposed method greatly improved the throughput of the compressor. *Media*, *comp*, and *repeat* showed 39.6%, 28.7%, and 60.2% throughput improvements, respectively. Much of *comp* and *media* were compressed in SBM. In the case of *repeat*, almost all files were compressed in SHM. Therefore, the compressibility prediction technique reduces the execution time consumed by Huffman code generation, which increases the throughput of the compressor.

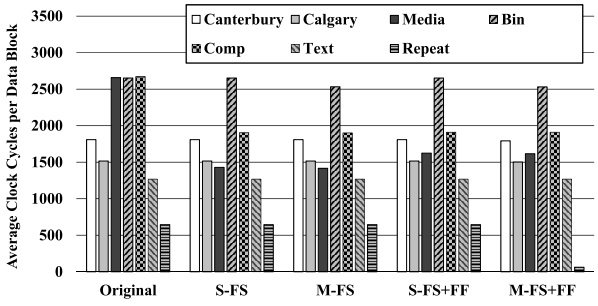
As shown in Fig. 21b, the compressibility prediction technique reduced the execution time of the Huffman code generator for processing *media*, *comp*, and *repeat* by 39.2%, 28.5%, and 90.5%, respectively.

Additionally, according to Fig. 21c, the number of clock cycles for which each module in the compressor was stalled based on processing delays in the other modules was reduced by 93.1%, 88.9%, and 99.4%, respectively. These two sets of experimental results suggest that the proposed method increases the throughput of the compressor by reducing the amount of computation that the Huffman code generator handles. *Normal* and *txt* showed no increase in compressor throughput because most of the files in these categories were compressed in DHM.

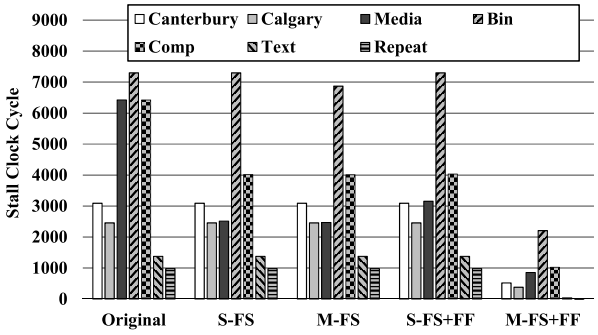
Compressibility prediction using only a few features may result in performance degradation of the compressor. In the cases of *normal* and *repeat*, the throughput of the compressor decreased by up to 0.003% and 0.28%, respectively. When



(a) Compression throughput with each entropy filtering scheme



(b) Average clock cycles consumed by Huffman code generator for processing a DB



(c) Clock cycles for which each module was stalled by processing delays

FIGURE 21. Compression throughput with each entropy filtering scheme.

compressing *media*, the throughput tends to decrease as the prediction accuracy increases. This is because as the accuracy of prediction increases, the percentage of DBs processed in SBM decreases.

F. RESOURCE CONSUMPTION

We analyzed LUT usage by synthesizing a DEFLATE compressor to evaluate the proposed technique in terms of resource consumption. First, we evaluate the effectiveness of the two hardware cost-reduction techniques. Fig. 22 shows the hardware resource consumption of compressibility predictor (i.e., *CP*) and selective compressor (i.e., *SC*) when utilizing the computational reuse technique (i.e., *CRT*) and the computation complexity reduction technique (i.e., *CCR*). As shown in the figure, two proposed techniques reduce the hardware resource consumption of CP and SC by 82.34% and 10.15%, respectively.

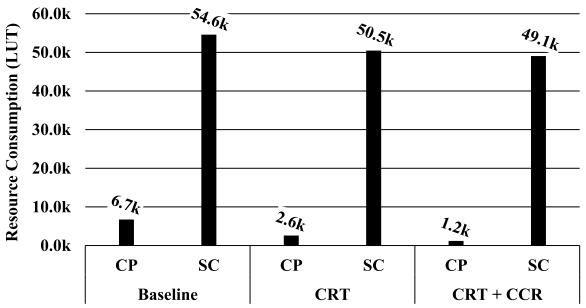


FIGURE 22. Hardware resource consumption of compressibility predictor and selective compressor when using proposed cost-reduction techniques.

TABLE 3. Total LUT consumption.

	Baseline	Proposed
Total	47869	49057
Feature-extraction	-	$402 \times N + 18 \times M$
Compressibility prediction	-	$366 \times M$

(#LZ77 encoder=N, #Huffman code generator=M)

Table 3 shows the hardware resources of the FPGAs that were consumed by each compressor before and after applying the proposed technique. The proposed compressor consumes approximately 3.1% more hardware resources than the baseline compressor. Additionally, Table 3 also shows the amount of hardware resources consumed by each hardware module for feature-extraction and compressibility prediction, respectively. The feature-extraction module operates in parallel with the LZ77 encoder. Additionally, the compressibility prediction module operates simultaneously with the Huffman code generator. Therefore, as the number of LZ77 encoders and Huffman code generators increases, the amount of hardware resources consumed by each module increases.

IX. CONCLUSION

This paper presented a low-cost compressibility prediction technique for high-performance lossless compression. We described an input DB analysis method for data compressibility prediction. The proposed method determines the compression mode of a DB during data compression based on the analysis results. Because the determination of the compression mode is faster than the actual compression process for each mode, the prediction method reduces the amount of computational resources wasted for compressing uncompressed data. It also improves the overall performance of the compressor by efficiently utilizing the computational resources. Experimental results demonstrated that the proposed method improves compression engine throughput by approximately 34.15%. Our analysis indicated a compression ratio loss of approximately 0.17% and an increase in hardware resource usage of approximately 3.1%, which are negligible.

ACKNOWLEDGMENT

The authors are grateful to the anonymous reviewers for their valuable feedback and comments.

REFERENCES

[1] U. Sivarajah, M. M. Kamal, Z. Irani, and V. Weerakkody, "Critical analysis of big data challenges and analytical methods," *J. Bus. Res.*, vol. 70, pp. 263–286, Jan. 2017.
[2] P. Barnaghi, A. Sheth, and C. Henson, "From data to actionable knowledge: Big data challenges in the Web of things," *IEEE Intell. Syst.*, vol. 28, no. 6, pp. 6–11, Nov./Dec. 2013.
[3] J. Chen, Y. Chen, X. Du, C. Li, J. Lu, S. Zhao, and X. Zhou, "Big data challenge: A data management perspective," *Frontiers Comput. Sci.*, vol. 7, no. 2, pp. 157–164, Apr. 2013.
[4] M. A. Vasarhelyi, A. Kogan, and B. M. Tuttle, "Big data in accounting: An overview," *Accounting Horizons*, vol. 29, no. 2, pp. 381–396, Jun. 2015.
[5] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[6] Y. Chen, A. Ganapathi, and R. H. Katz, "To compress or not to compress–compute vs. IO tradeoffs for mapreduce energy efficiency," in *Proc. 1st ACM SIGCOMM workshop Green Netw. (Green Netw.)*. New York, NY, USA: ACM, 2010, pp. 23–28.
[7] *Aws Global Infrastructure*. Accessed: Sep. 27, 2019. [Online]. Available: https://aws.amazon.com/ko/about-aws/global-infrastructure/
[8] T. Davenport and J. Harris, *Competing on Analytics: Updated, with a New Introduction: The New Science of Winning*. Brighton, MA, USA: Harvard Business Press, 2017.
[9] C. Mullins. (1999). *Database Trends*. [Online]. Available: http://www.bwdb2ug.org/Presentations/DatabaseTrends.pdf
[10] M. Atkinson, *Uk e-Science Grid Infrastructure Meets Biological Research Challenges*, document, 2002.
[11] H. Liu and D. Orban, "GridBatch: Cloud computing for large-scale data-intensive batch applications," in *Proc. 8th IEEE Int. Symp. Cluster Comput. Grid (CCGRID)*, May 2008, pp. 295–305.
[12] D. Harnik, R. I. Kat, O. Margalit, D. Sotnikov, and A. Traeger, "To zip or not to zip: Effective resource usage for real-time compression," in *Proc. FAST*, 2013, pp. 229–242.
[13] P. A. H. Peterson and P. L. Reiher, "Datacomp: Locally independent adaptive compression for real-world systems," in *Proc. IEEE 36th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2016, pp. 211–220.
[14] T. Summers and S. A. Engineer, "Hardware based gzip compression, benefits and applications," *CORPUS*, vol. 3, no. 2.75, pp. 2–68, 2008.
[15] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok, "Energy and performance evaluation of lossless file data compression on server systems," in *Proc. Israeli Experim. Syst. Conf. (SYSTOR)*. New York, NY, USA: ACM, 2009, pp. 4:1–4:12.
[16] B. Nicolae, "High throughput data-compression for cloud storage," in *Proc. 3rd Int. Conf. Data Manage. Grid Peer-To-Peer Syst.* Berlin, Germany: Springer-Verlag, 2010, pp. 1–12.
[17] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proc. 42nd Annu. Int. Symp. Comput. Archit. (ISCA)*. New York, NY, USA: ACM, 2015, pp. 158–169.
[18] D. C. Marinescu, *Cloud Computing: Theory and Practice*. San Mateo, CA, USA: Morgan Kaufmann, 2017.
[19] K. Soo Yim, H. Bahn, and K. Koh, "A flash compression layer for SmartMedia card systems," *IEEE Trans. Consum. Electron.*, vol. 50, no. 1, pp. 192–197, Feb. 2004.
[20] Y. Park and J.-S. Kim, "ZFTL: Power-efficient data compression support for NAND flash-based consumer electronics devices," *IEEE Trans. Consum. Electron.*, vol. 57, no. 3, pp. 1148–1156, Aug. 2011.
[21] W.-T. Huang, C.-T. Chen, Y.-S. Chen, and C.-H. Chen, "A compression layer for NAND type flash memory systems," in *Proc. 3rd Int. Conf. Inf. Technol. Appl. (ICITA)*, vol. 1, Jul. 2005, pp. 599–604.
[22] B. Nicolae, "On the benefits of transparent compression for cost-effective cloud data storage," in *Transactions on Large-Scale Data- and Knowledge-Centered Systems III*. Springer, 2011, pp. 167–184.

[23] *Btrfs Compression, Transparent File Compression*. Accessed: Sep. 30, 2019. [Online]. Available: https://btrfs.wiki.kernel.org/index.php/Compression
[24] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, "ZBD: Using transparent compression at the block level to increase storage space efficiency," in *Proc. Int. Workshop Storage Netw. Archit. Parallel I/Os*, May 2010, pp. 61–70.
[25] R. Filgueira, M. Atkinson, Y. Tanimura, and I. Kojima, "Applying selectively parallel I/O compression to parallel storage systems," in *Proc. Eur. Conf. Parallel Process.* Springer, 2014, pp. 282–293.
[26] *DB-Engines Database Management Systems Popularity Ranking*. Accessed: Sep. 30, 2019. [Online]. Available: http://db-engines.com/en/ranking
[27] J. Zhang, *Real-Time Lossless Compression of SoC Trace Data*, document, 2015.
[28] P. Deutsch and J.-L. Gailly, *ZLIB Compressed Data Format Specification Version 3.3*, document RFC 1950, 1996.
[29] *Genomic Data 101: 2018 Edition, an Introduction to Genomic Data*, Front Line Genomics Mag., 2018.
[30] Petagene. *Transparent Lossless Compression*. [Online]. Available: https://www.petagene.com/cost-calculator/
[31] C. Burns, B. Tuv-El, J. Quintal, and J. Tate, *IBM Real-time Compression in IBM SAN Volume Controller and IBM Storwize V7000*. 2015.
[32] R. Tretau, J. Kim, B. Nolte, G. Nunn, and F. Schneider, *Introduction to IBM Real-time Compression Appliances*. IBM Redbooks, 2013.
[33] G. Pekhimenko, C. Guo, M. Jeon, P. Huang, and L. Zhou, "TerseCades: Efficient data compression in stream processing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2018, pp. 307–320.
[34] A. Martin, D. Jamsek, and K. Agarawal, "Fpga-based application acceleration: Case study with GZIP compression/decompression streaming engine," in *Proc. ICCAD Special Session 7C*, 2013.
[35] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, "FPGA implementation of GZIP compression and decompression for IDC services," in *Proc. Int. Conf. Field-Program. Technol.*, Dec. 2010, pp. 265–268.
[36] M. S. Abdelfattah, A. Hagiescu, and D. Singh, "Gzip on a chip: High performance lossless data compression on FPGAs using OpenCL," in *Proc. Int. Workshop OpenCL (IWOCL)*. New York, NY, USA: ACM, 2014, p. 4.
[37] 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.
[38] J. Matai, J.-Y. Kim, and R. Kastner, "Energy efficient canonical Huffman encoding," in *Proc. IEEE 25th Int. Conf. Appl.-Specific Syst., Archit. Processors*, Jun. 2014, pp. 202–209.
[39] W. Qiao, J. Du, Z. Fang, M. Lo, M.-C.-F. Chang, and J. Cong, "High-throughput lossless compression on tightly coupled CPU-FPGA platforms," in *Proc. IEEE 26th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*. New York, NY, USA: ACM, Apr. 2018, p. 291.
[40] S. Rigler, W. Bishop, and A. Kennings, "FPGA-based lossless data compression using Huffman and LZ77 algorithms," in *Proc. Can. Conf. Electr. Comput. Eng.*, 2007, pp. 1235–1238.
[41] 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.
[42] (2015). *Aha374/Aha378 PCI Express Compression and Decompression Accelerator Card*. [Online]. Available: http://www.aha.com/Uploads/aha374-378_brief_rev_c1.pdf
[43] *Dx2040, High Performance Scalable Solutions for Data Analytics, Storage, and Networking*. [Online]. Available: https://www.exar.com/content/document.ashx?id=21618
[44] A. El-Shimi, R. Kalach, A. Kumar, A. Oltean, J. Li, and S. Sengupta, "Primary data deduplication-large scale study and system design," in *Proc. USENIX Conf. Annu. Tech. Conf. (USENIX ATC)*. Berkeley, CA, USA: USENIX Association, 2012, p. 26.
[45] A. Kattan and R. Poli, "Genetic-programming based prediction of data compression saving," in *Proc. Int. Conf. Artif. Evol. (Evol. Artificielle)*. Springer, 2009, pp. 182–193.
[46] W. Culhane, "Statistical measures as predictors of compression savings," Ph.D. dissertation, Ohio State Univ., Columbus, OH, USA, 2008.
[47] D. A. Owusu, "Modeling outputs of efficient compressibility estimators," Ph.D. dissertation, Univ. Minnesota, Minneapolis, MN, USA, 2018.
[48] 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.
[49] 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. C-66, no. 7, pp. 1188–1199, Jul. 2017.
[50] 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.
[51] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proc. IRE*, vol. 40, no. 9, pp. 1098–1101, Sep. 1952.
[52] D. Salomon, *A Concise Introduction to Data Compression*. London, U.K.: Springer-Verlag, 2008.
[53] C. E. Shannon, "A mathematical theory of communication," *Bell Syst. Tech. J.*, vol. 27, no. 3, pp. 379–423, Jul./Oct. 1948.
[54] R. Lyda and J. Hamrock, "Using entropy analysis to find encrypted and packed malware," *IEEE Secur. Privacy Mag.*, vol. 5, no. 2, pp. 40–45, Mar. 2007.
[55] G. Frantz and R. Simar, *Comparing Fixed- and Floating-Point DSPS*. Dallas, TX, USA: Texas Instruments, 2004.
[56] R. Gutierrez and J. Valls, "Low cost hardware implementation of logarithm approximation," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 12, pp. 2326–2330, Dec. 2011.
[57] S. Hashemi, R. I. Bahar, and S. Reda, "A low-power dynamic divider for approximate applications," in *Proc. 53rd Annu. Des. Autom. Conf. (DAC)*. New York, NY, USA: ACM, 2016, p. 105.
[58] Matt Powell. (2001). *The Canterbury Corpus*. [Online]. Available: http://corpus.canterbury.ac.nz/descriptions/
[59] Marie Lebert. *Project Gutenberg*. [Online]. Available: http://www.gutenberg.org/
[60] *Core.2 Global Air-Sea Flux Dataset*. [Online]. Available: https://rda.ucar.edu/datasets/ds260.2/
[61] *NCEP GDAS/FNL 0.25 Degree Global Tropospheric Analyses and Forecast Grids*. [Online]. Available: https://rda.ucar.edu/datasets/ds083.3/
[62] K. Conrad, *Probability Distributions and Maximum Entropy*.

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.

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.

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

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 from the Department of Electronics and Computer Engineering, Hanyang University, Seoul, in 2017. He is currently pursuing the Ph.D. degree with the Department of Electronics Engineering, Hanyang University.

His research interests include embedded systems and nand flash memories.

JOONYONG JEONG received the bachelor's degree in information systems from Hanyang University, where he is currently pursuing the integrated M.S./Ph.D. degree in electronics and computer engineering. His research interests include computer architecture, big data, and key value store systems.

JAEWOOK KWAK received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2015. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University.

His research interests include computer architecture, embedded systems, and nand flash-based storage systems.

JUNGKEOL LEE received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2014. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University.

His research interests include embedded computing and the IoT devices.

GYEONGYONG LEE received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2014. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University.

His research interests include embedded computing and nand flash memories.

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

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, Hanyang University, in 2015. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University.

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

JINWOO JEONG received the B.S. degree from the Department of Electronic Engineering, Hanyang University, South Korea, in 2015. He is currently pursuing the Ph.D. degree with the Department of Electronics and Computer Engineering, Hanyang University.

His research interests include nand flash-based storage systems and error-correction codes.

WANG KEXIN (Student Member, IEEE) received the B.S. degree from the School of Optoelectronic Engineering, Changchun University of Science and Technology, in 2017. She is currently pursuing the master's and Ph.D. degrees with the Department of Electronic and Computer Engineering, Hanyang University, Seoul, South Korea.

Her research interest is in high-performance solid-state drive (SSD) architecture.

YONG HO SONG (Member, IEEE) 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 Electronics Engineering, Hanyang University, Seoul, and the 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.

Dr. Song has served as the Program Committee Member for 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.