electronics

MDPI

Article

Exception Handling Method Based on Event from Look-Up Table Applying Stream-Based Lossless Data Compression

Shinichi Yamagiwa [1,2,*], Koichi Marumo [1] and Suzukaze Kuwabara [3]

[1] Faculty of Engineering, Information and Systems, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan; marumo@padc.cs.tsukuba.ac.jp
[2] JST, PRESTO, 4-1-8 Honcho, Kawaguchi, Saitama 332-0012, Japan
[3] Department of Computer Science, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan; kuwabara@padc.cs.tsukuba.ac.jp
* Correspondence: yamagiwa@cs.tsukuba.ac.jp

Abstract: It is getting popular to implement an environment where communications are performed remotely among IoT edge devices, such as sensory devices and the cloud servers due to applying, for example, artificial intelligence algorithms to the system. In such situations that handle big data, lossless data compression is one of the solutions to reduce the big data. In particular, the stream-based data compression technology is focused on such systems to compress infinitely continuous data stream with very small delay. However, during the continuous data compression process, it is not able to insert an exception code among the compressed data without any additional mechanisms, such as data framing and the packeting technique, as used in networking technologies. The exception code indicates configurations for the compressor/decompressor and/or its peripheral logics. Then, it is used in real time for the configuration of parameters against those components. To implement the exception code, data compression algorithm must include a mechanism to distinguish original data before compression and the exception code clearly. However, the conventional algorithms do not include such mechanism. This paper proposes novel methods to implement the exception code in data compression that uses look-up table, called the exception symbol. Additionally, we describe implementation details of the method by applying it to algorithms of stream-based data compression. Because some of the proposed mechanisms need to reserve entries in the table, we also discuss the effect against data compression performance according to experimental evaluations.

Keywords: lossless data compression; exception code; stream-based lossless data compression; LCA-DLT; ASE coding

Citation: Yamagiwa, S.; Marumo, K.; Kuwabara, S. Exception Handling Method Based on Event from Look-Up Table Applying Stream- Based Lossless Data Compression. *Electronics* **2021**, *10*, 240. https://doi.org/10.3390/electronics10030240

Received: 12 December 2020
Accepted: 19 January 2021
Published: 21 January 2021

1. Introduction

Due to rapid performance improvement of information processing equipment such as recently IoTsystems, the amount of data used in the system is still increasing fast in the cases that big data applications such as algorithms of artificial intelligence are employed. Focusing on data communication between the edge devices and the cloud servers or data migration among the processing components in the equipment, the recent performance growth of the physical communication data path is not so high than the one of producing data from equipment. For example, video resolution is growing at four times almost every year from HD to 8 K. In addition, the motion sensory devices output its data in 32 bit wide since it was eight bit wide in several years before. In this paper, we focus on techniques to reduce the amount of data by employing data compression technology.

Data compression technologies are categorized as *lossless* and *lossy* ones. The former decodes the compressed data to the original data without any loss. The latter does not maintain the original data. It mainly treats multimedia data by removing insensible data in high/low frequency of such as images, sensory data and sound waves of pulse code

modulation. The MPEG format is well-known. In this paper, however, we focus on the lossless compression.

We developed stream-based data compression methods that are based on the *digram coding*. The digram coding uses a look-up table that is used to compress an original data with associating to the index. The methods, called LCA-DLT [1] and ASE coding [2], employ a fixed number of entries in the table and recycle the entries during the compression/decompression. LCA-DLT organizes a compressor/decompressor module that converts a pair of original data to/from an index of the look-up table. The modules can be connected to compress/decompress multiple data pairs. LCA-DLT can be easily implemented in hardware and the resource size is selectable depending on the number of connected modules. However, because the compressed data is associated from the indices of the look-up table and the size of the index is fixed, the method results low dynamicity in the aspect of the compression ratio. ASE coding improves this problem. It compresses an original data to a short bit sequence by an instantaneous entropy calculation from the number of occupied entries in the look-up table. This follows entropy of data stream in real time and results high dynamicity of the compression ratio. ASE coding also can be compactly implemented on FPGA working at 250 MHz.

Employing the compressor in the producer of data stream and the decompressor in the consumer, the stream-based data compression reduces the latency and increases the bandwidth in the communication data path. This brings performance upgrade of the physical media virtually. In particular, under the condition where an application that needs fast response from the consumer of the data such as IoT systems, this helps to improve overall performance of the system. However, when we consider to generate an exceptional procedure without related to the compression process such as a reconfiguration of the decompressor's setting from the producer side, we need to compress all data including the exception code. Then, after the decompression, the original data must be analyzed based on the data format of the original data. On the other hand, if we try to embed the exception code as a compressed data, by assigning reserved bit patterns, the compressor is not able to distinguish the exception code from the compressed data because all combinations of available bit patterns would appear in original data. Thus, we are not able to embed exception code in a part of compressed data stream without any additional mechanism.

This paper will propose novel methods to handle the exception code focusing on data compression algorithms with a fixed number of look-up table entries. As typical examples, we employ the proposed method on the stream-based ones: LCA-DLT and ASE coding. The proposed methods are also available in the conventional digram codings. In this paper, we will also evaluate the performance tradeoffs by applying the methods on those stream-based ones.

The main contributions of research results disseminated in this paper are:

- Designs and implementations of handling methods for the exception code applying to the digram coding with a fixed number of entries in look-up table.
- Effective classification of the proposed method for handling the exception code by categorizing into four classes (RETO, RETI, FETI and FETO).
- Implementation details of the classes and evaluations of the compression performance.

This paper is organized as follows. The next section will describe the backgrounds and definition of this research regarding the data compression technologies and the exception handling methods. Section 3 will explain the target system model. In the system, we will introduce situations wherein the exception code is required. Then, we will show the proposed method to handle the exception code in data compression algorithms with a fixed number of entries in the look-up table. Section 4 will show performance evaluations focusing on the effects when the methods are applied. Finally, we will conclude the research results.

2. Backgrounds and Definitions

2.1. Stream-Based Lossless Data Compression

The recent type of data produced from equipment of information technology such as IoT devices mainly forms a continuous data stream such as video images and sensor data [3]. The speed and the amount of the data stream are increasing rapidly. To support the technological aspects of the equipment, we need to reduce the amount of data itself. Therefore, we focus on lossless data compression technology that can reduce the data amount of the data stream without any quality loss. To implement compression methods for data streams, we need to address the following factors in the algorithm: (1) the algorithm does not need any buffering during compression/decompression processes. (2) it compresses/decompresses a unit of data (we call this a *symbol*) in a small delay. (3) it is implementable in hardware compactly and works fast to support performance of the physical media. Due to the algorithm that supports these factors, the stream-based lossless data compression implements fast compression based on hardware and brings compactization of systems.

Lossless data compression technology has its origins in Shannon's entropy since the 1950s. The method assigns S bits to each symbol of an original data sequence where S is an entropy represented by $-\sum p \log_2 p_i$ where it is calculated from frequent probability of each symbol p_i. Then, arithmetic coding [4,5] is the next generation of compression algorithm. It represents the target data by numeric values. It assigns each data pattern to a value in a domain where includes all symbols are presented. This mechanism improves the compression ratio, better than Shannon's entropy. However, it needs to process all input data and must decide the domain to express all symbols appeared in the input data. The algorithm tries to find the frequent information of all symbols in the input data. The Range Coder [6] is a similar compression algorithm to the arithmetic coding. Huffman coding [7] is another lossless data compression. It also has the disadvantage of the buffering problem against processing data stream because it needs to create a binary tree of whole data. To overcome it, dynamic Huffman coding [8] was proposed. It arranges a binary tree dynamically created during the compression process. However, it is too heavy calculation to process very fast data stream. Furthermore, it is too complicated to implement it on fast and small hardware. Therefore, the algorithms mentioned above are not suitable to process data stream.

In the 1970s, an algorithm based on look-up table that registers frequent symbols was proposed. A typical implementation was LZW (Lempel-Ziv-Welch) [9,10] instantiated on Zip, LZ4 [11], Snappy [12], and deflate [13]. The algorithm compresses one or more frequent symbols by using a look-up table. The look-up table maintains the frequent symbol patterns and the compressed symbols are translated as the table indices. This kind of algorithm that uses a look-up table is called *digram coding*. For example, Figure 1 shows examples of compression/decompression processes of LZW. It compresses an ASCII character pattern "ABCABC". In LZW, first, the table is initialized by available symbols appearing in the patterns. Typically, a table entry has a 12 bit symbol and the first 256 entries are configured to the corresponding binary bits. The compressor has a rule that a new pattern with the subsequent symbols is added to the table. While a pattern is hit in the table, the associated index is outputted. Here, to explain simply, the example initializes the table with 'A', 'B' and 'C' first as shown in Figure 1a. The first 'A' hits in the table. Then, "AB" is tried if it is hit or not in the table. However, it misses. Therefore, 'A' is translated to the index '0' with $ceil(\log_2 k)$ bits where k is the maximum number of entries in the table. The pattern "AB" is added to the table. The 'B' is converted to '1' as depicted in Figure 1b. With the similar operations, "ABC" is added to the table and the compressor outputs '2' as shown in Figure 1c. After that, as illustrated in Figure 1d, "ABC" is hit in the table and the compressor outputs '5'. This results "0125". If the table index is 3 bits, the original data pattern is compressed from 48 bits to 12 bits. In the decompressor, the table is initialized as well as the compressor. During accepting the indices of compressed data, the decompressor

adds new patterns to the table as well as the compressor. The compressed pattern can be decompressed to 'A', 'B', 'C', and "ABC".

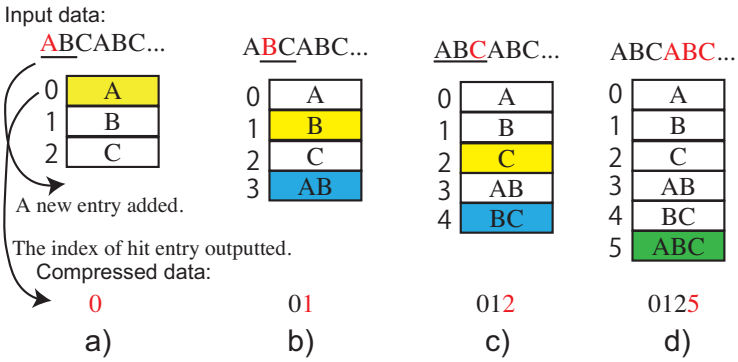

Figure 1. Compression processes of LZW. (**a**) shows an initial state of the table; (**b**) and (**c**) illustrate the ones after "AB" and "BC" are added, respectively; (**d**) shows the one after "ABC" is added.

As we can see above, the algorithm can process data stream. However, it has a fatal drawback that the memory size for the look-up table increases infinitely as the number of symbols are compressed. This means we cannot know the required amount of memory during the compression. Because we cannot determine a fixed number of hardware resources used in the conventional digram coding such as LZW, we need to give some restrictions to the algorithm. For example, in LZW case, we can estimate the maximal size of memory for the look-up table. However, we cannot implement the algorithm on any fixed amount of memory by assigning a limited number of entries in the table. Therefore, it is typically configured to a fixed number of table entries such as 4096 used in the implementation [14]. This is controlled in the algorithm and resets the table to process the subsequent data from the initialized table. Therefore, it is not suitable for the implementation that limits available resources such as hardware. Thus, the conventional digram coding does not satisfy the conditions of stream-based data compression because the perfect implementation needs unpredictable amount of resources.

On the other hand, we developed stream-based data compression algorithms. First, we developed a very simple one called LCA-SLT [15] that supports the concept of LCA [16]. It prepares a look-up table with several entries. Each entry maintains a symbol pair (s_0, s_1). When a symbol pair hits in the table, the compressor translates it to the table index I. When a mishit happens, it outputs the original symbol pair. Each output combined with a *CMark* bit that indicates if it is a compressed or an original symbol. Initially, entries are set to the pairs statistically selected from a sample data sequence. This is suitable for hardware implementation because the number of processing steps are constant. We can implement the algorithm in a small and fast hardware and also can achieve high bandwidth. Additionally, connecting the compressors, LCA-SLT can also compress a long pattern. However, due to the static table associated from the sample data sequence, when the frequency of patterns (i.e., data entropy) in a data stream changes, the compression ratio is affected significantly. To improve the compression performance, we developed LCA-DLT that includes entry exchange mechanism in the look-up table.

We added a dynamic histogram control to the look-up table of LCA-SLT and proposed LCA-DLT [17]. When a symbol pair is received by the compressor, it is registered in an entry of the table when the compressor/decompressor does not find it in the table. We also prepared a reference counter in every table entry. The counter is reset to an initialized value at a registration of a symbol pair. We also prepared a remove pointer that rounds the table entries. The counter is decremented when the entry is pointed. Then, if the counter is zero, the entry is invalidated and recycled. At every table search operation, the reference counter is decremented at a mishit. On the other hand, at a hit, the counter is incremented. Therefore, frequent symbol pairs can be maintained due to this counter operation. Figure 2 shows an example of the compression processes of an ASCII data pattern. Figure 2a shows

the initial registration of a symbol pair and outputs the original data. Actually, the output needs to combine a CMark bit (in this case, 0) to indicate that it is not compressed. Figure 2b shows a hit case when an input symbol pair is matched in the table. This case needs to add a CMark bit (here, it is 1). Note that the hit case increments the reference count value. Then, it is incremented when the entry is used again. In this example, a compressed symbol becomes three bits due to four entries in the table. When a data pair is not compressed, the output becomes 17 bits. After processing the pairs, Figure 2c shows an example when the entries in the table are invalidated. When the counter becomes zero, the entry is invalidated and reused for the subsequent input symbol pairs. The decompressor processes the same operations as the compressor does with reading the CMark bit at every output from the compressor. During the registration operation for a new entry to the table, any entry might not be invalidated. In this case, we can use the lazy compression mechanism [1] to skip the symbol pair. As explained above, LCA-DLT compresses data stream without stalling by using a fixed number of entries. Therefore, it is easily implemented on hardware with a fixed amount of resources. In [18], we reported an implementation with 200 MHz on an FPGA device.

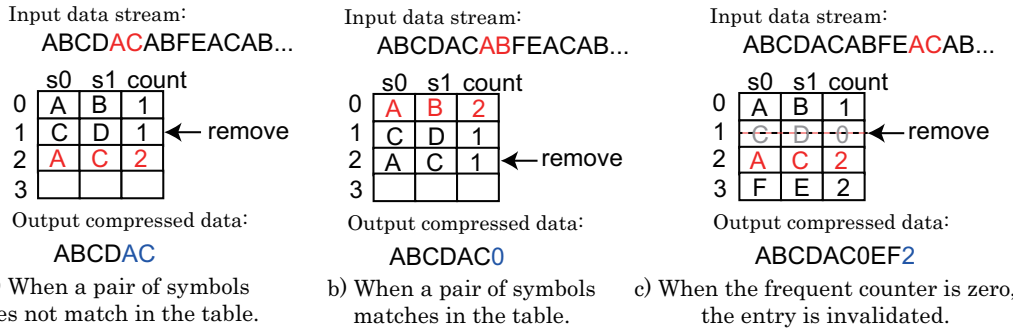

Figure 2. Example of compression processes of LCA-DLT.

The compressed data by the compressor of LCA-DLT consists of the full bits of the look-up table. This rises a compression limit that the compression ratio is affected by the number of entries in the look-up table. To overcome this performance limit, we developed a new algorithm called ASE coding [2]. It compresses a symbol using an effect that the data entropy of data stream follows the number of occupied entries in the look-up table as used in LCA-DLT. The instantaneous compressed data is shrunk to m bits from the number of occupied entries k by an entropy calculation of the equation $m = ceil(\log_2 k)$. Figure 3 shows an example of the compression/decompression mechanism. Figure 3a illustrates that the compressor outputs a compressed data by shrinking the table index when the input symbol is matched in the table. The number of bits in the compressed data becomes $m + 1$ bits due to a CMark bit. When the decompressor receives the compressed data, it reads the CMark bit. If it is set, the compressor calculates m, extracts m bits from the compressed data stream and picks up the original symbol from the table. If the input symbol does not match in the table, the compressor registers the symbol to the table and outputs the original symbol with a CMark bit (in this case, 0). When the decompressor receives the CMark bit, it can know if it is compressed or not. If compressed, the subsequent m bits are extracted by the entropy calculation and are extended to the number of bits corresponding to the table index as depicted in Figure 3a. If not, the decompressor extracts the number of bits of an original symbol and registers it to the table. During the compression/decompression operations, the table is operated by the method shown in Figure 3b. When the input symbol is hit in the table, the entry is moved to the top and the others are pushed to the next ones such as the LRU (Least Recently Used). When a mishit happens, the input symbol is pushed from the top entry in the table. However, while repeating the registration operations, the table always becomes full. This results in full bits of the table index as the compressed data. In this case, any original symbol is not compressed. To avoid this situation, we apply *entropy culling* that invalidates the highest entry in the table after several hit operations.

By performing these operations, ASE coding implements a stream-based compression mechanism by using a fixed number of entries in the table.

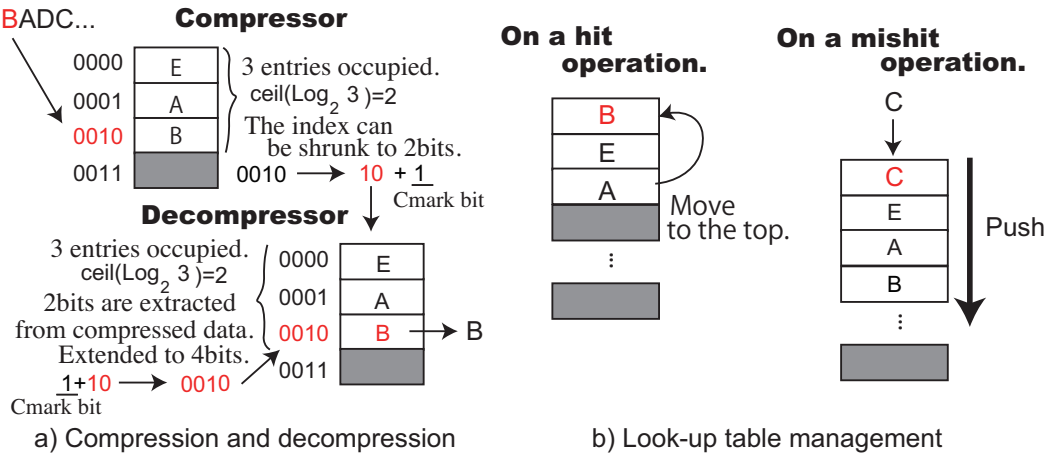

Figure 3. Example of compression processes of ASE coding.

As explained above, ASE coding compresses the data stream without stalling the flow. It is also easily implemented on hardware due to the simple compression mechanism. The hardware resource size is deterministic because the number of entries in the look-up table is fixed. ASE coding can be implemented on hardware by a smaller resource size than LCA-DLT and also works at the same speed on FPGA devices. Thus, we implemented an effective lossless data compression targeted to data stream.

According to the discussion above, LCA-DLT and ASE coding support the conditions for the stream-based lossless data compression: (1) compression process without buffering the input data, (2) compression/decompression processes in a small delay and (3) compact hardware implementation. Thus, these compression algorithms provide a breakthrough to overcome the enormous increasing of the big data processing demands under the fast data communication situation.

2.2. Exception Handling on Lossless Data Compression

In information equipment with a stream-based data compression mechanism through the data path, the compressed data stream is continuously transferred from the producer to the consumer of the data stream. During the continuous data flow, the system needs to allow exchanging controls for the peripheral logics and/or the compression algorithm itself. The stream-based lossless data compression can control and adjust the compression performance against the target data in real time if it can send the compressed data with the control data in a single stream. For example, the digram coding will become available to reset the look-up table and also to adjust parameters for the backend system of the decompression algorithm such as sensor settings for multimedia application. When the compressor and the decompressor are implemented in hardware, the parameter adjustment must be performed in real time during the processes of a data stream. To implement this mechanism, we need to add a method to the digram coding. The method embeds an exception code that is distinguished from the normal compressed data. In the other words, the exception code is inserted in compressed data, and then, the decompressor must recognize the code seamlessly. Thus, we need to invent a mechanism to insert a control code under an exception status. In this paper, we call this control code an *exception symbol*.

First, the simplest method to embed exception symbols in a compressed data stream is to compress its original data stream with the exception symbols. Figure 4a shows an example of the method. It passes a data stream to the compressor that is formatted in a rule. The exception symbols are recognized in the decompressor side after decompressed to the original data. This method has a drawback that the original data needs additional information for distinguishing data among the compressed ones and exception symbols.

Furthermore, when we consider to configure the compressor and the decompressor settings via the exception symbols, it is hard to synchronize the configuration timings because the timings must be adjusted according to the delay of the compressor/decompressor.

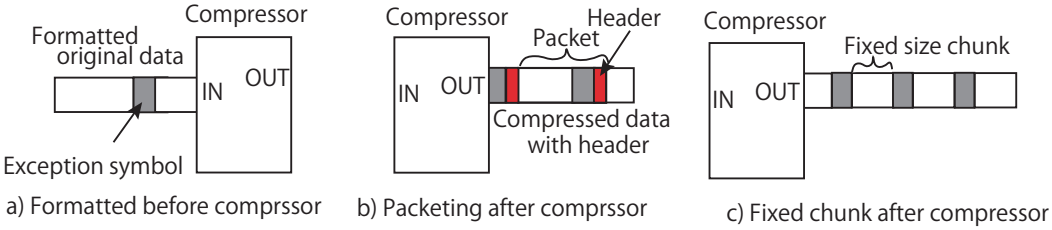

Figure 4. The simplest methods for exception symbol treatment on digram coding.

The second method is to make packets of compressed data and exception symbols. This method needs to add a header information to each chunk. Figure 4b illustrates an example of this method. This method needs to include a communication protocol such as that used in a packet-based network connection. This demands the compressor/decompressor to include a protocol. The compressor specifies a protocol to create a packet with chunks. Then, the decompressor extracts the chunks from the packet by following the protocol. This obviously degrades the compression performance due to the additional information combined with the compressed data stream according to the packet format. Similarly, the escape code is another well-known method that distinguishes the exception symbol and the compressed data. For example, if the code of 'FF' is defined as the escape code, the subsequent code is recognized as the exception symbol. Here, the compressed data that corresponds to the escape code is expressed two 'FF's. This means that the method inevitably increases the amount of the compressed data, and thus degrades the compression performance.

The third method is to make chunks from the compressed data separated in a fixed size. The compressor outputs those chunks and inserts the exception symbol between the chunks as illustrated in Figure 4c. This method avoids additional process in the compressor to format a packet like the second method. However, the exception symbol must be inserted after every chunk. Even if any exception does not occur, the exception symbol must be inserted by defining a void exception such as NOP (No OPeration). Although this mechanism can avoid equiping a protocol on both the compressor and the decompressor, the void exception increases the amount of compressed data. Therefore, the compression performance degrades. LZ4 [11] uses the method to make chunks and implements a pseudo stream-based compression manner. It can use this method for the exception symbol. However, it cannot avoid degrading the compression performance.

The three methods above to insert the exception symbol to a compressed data stream are applicable to the conventional lossless data compression methods such as Shannon's entropy and Huffman coding. However, when we consider to introduce the exception symbol to a digram coding, we must employ another method to support it by using the look-up table. For example, LZW implements the mechanism of exception symbol using the look-up table. It initially prepares entries in the table dedicated to exception symbols. At the initialization, the compressor and the decompressor prepare a look-up table respectively, setup the table with the available bit patterns for all single symbols and add two additional entries for *stop* and *clear* exceptions after the occupied entries. As a typical setting, initially 4096 entries are prepared in the table. The lower entries are initialized from 0 to 255 at first. The 257th and 258th entries are used to save the codes for the stop and the clear exceptions. The clear exception is used to reset the table. The stop exception is used to stop compression/decompression. The compressed data '256' and '257' are detected as the exception symbols in the decompression side. In this method, we need to prepare at least $2^k + n$ entries in the table, where k is the number of bits in an original symbol and n is the number of exception symbols. Therefore, the number of bits in a compressed data becomes larger than the original symbol because the number of bits

in the table index begins with $ceil(\log_2(2^k + n))$ bits. In the LZW case, two entries for the exception symbols are not used for the compression processes.

However, the method stated above for managing exception symbols is not applicable to the digram coding with a fixed number of entries because the table entries are reused by the invalidation mechanism dynamically during the compression algorithm. For example, LCA-DLT and ASE coding, explained in the previous section, do not have any mechanism to reserve the entries for the exception symbols without invalidation. Even when the additional entries reserved for the exception symbols are extended in the last of the table, the number of bits in the compressed symbol increases and the compression ratio will become worse. Thus, in the aspects of control and performance, digram coding with a fixed number of table entries is not able to use the similar method employed in LZW that reserves the exception symbols in the table.

Moreover, the method used in LZW needs to register all exception symbols in the look-up table. However, such as performed in the processor architecture field, the "exception" is an event to migrate the operation mode from a normal to a supervisor mode. The processor recognizes which exception has occurred from status information that causes the event. As well as this technique, just an exception symbol for the event that migrates the compressor's/decompressor's mode to a supervisor mode should be prepared in the data compression algorithm. In addition, the information by which exception is caused should follow the exception symbol. However, the conventional exception methods in lossless data compression do not support this kind of diversity.

As discussed in the sections above, the stream data compression is able to compress/decompress data stream without any stalls and can be implemented on hardware compactly. However, we do not have any smart solution to support the exception symbol that causes an event to migrate the operation mode using a look-up table with a fixed number of entries. To support this, we need a novel mechanism to cause an exception event in the fixed-size look-up table. In the rest of this paper, we will propose a new mechanism to support the exception symbol that can follow the additional information for the event without degradation of data compression performance.

3. Exception Handling Method for Stream-Based Lossless Data Compression

Let us propose a novel exception handling mechanism in the digram coding that maintains the look-up table with a fixed number of entries.

3.1. Modeling of Variable Exception Control

Here, we begin to define a system model with compressor/decompressor that accepts parameter configuration for the peripheral devices/algorithms. This system has requirements as follows:

- The number of entries in the look-up table is a constant. The number of entries is fix to a number decided initially. It is not changed dynamically.
- All entries in the table can be modified/moved/invalidated.
- Each table entry has a valid bit that indicates if the entry is occupied or not.
- The parameters of peripheral programs/logics around the compressor/decompressor can be modified independently.

To satisfy the conditions above, the compressor and the decompressor need to detect the exception symbol before beginning the compression and decompression process for a symbol, respectively. These mechanisms are needed for eliminating compressed data and exception symbol. The compressor has inputs for exception symbols. The decompressor has outputs for parameter updates. For example, when the exception symbol is requested to the compressor by asserting the enable input for an exception, the compressor outputs an exception symbol as a compressed data. When the exception symbol arrives at the decompressor, it detects the exception symbol and asserts the parameter updates. In order to synchronize the parameter update timings at both the compressor and the decompressor, we need to serialize inputted data to be compressed and exception symbols. When a

hardware implementation, during compression of target data, the compressor stops the input in order to give a timing to insert an exception symbol.

Furthermore, although the compressed data and the exception symbols are merged in the data stream outputted from the compressor, the decompressor must exploit the exception symbols obviously. Additionally, the exception symbols are generated from the compressor without storing them in entries of the look-up table and report the occurrences of the corresponding exceptions to the decompressor. In this case, this mechanism can embed *command codes* after the exception symbol. The command codes are translated by an identified protocol defined in the system. The command code mechanism is also available in the method of LZW. When the exception symbol is received in the decompressor, it can recognize the command codes after the exception symbol by receiving the subsequent data in the compressed data.

As discussed above, if we can implement a method to insert the exception symbols into compressed data at precise timings for updating parameters in peripherals, we will be free from timing conditions to implement compression/decompression hardware that has strict timing restrictions. Moreover, we can freely define command sequences with arguments by the command code regarding its exceptions.

The peripheral logics/algorithms control the compressor. By inserting the exception symbols, those exchange the control code and the parameters for the application in the system. In particular, the exception symbol will address the following situations in the stream-based data compression:

- The exception symbol implements in order to insert *pause* function in the stream-based compression. For example, during the continuous compression process for a data stream, if the peripheral logic/algorithm that generates the data stream wants to stop the generation for a while, the conventional digram coding is not able to implement the pause function. Because the compressor is not able to define special code for the compressed data, the decompressor is not able to know the timing to ignore the part of data stream. The exception symbol will address this situation to notify the decompressor to ignore the compressed data generated by the compressor during the *pause* period by an exception symbol.
- It is not possible to know the end of data stream (EOS) when the decompressor receives a limited length of data stream. For example, an application needs to decompress a data file compressed by a stream-based data compressor. If it is performed by a software decompressor, the file size is known. However, if it is implemented in hardware, due to the pipeline of the decompression processes, it is impossible to know the *stop* timing of the decompression because it already processes a part of the decompression operation in the pipeline. When the multiple files are decompressed in the same manner, the situation becomes worse. The decompressor does not have method to know the borders of streams from different files. The exception symbol will address the EOS. Thus, the decompressor can know it in the precise timing before the decompression pipeline.
- One of typical realistic applications that can apply the exception symbol and the command codes are found in the communication flow control between a data producer and its consumer. For example, as mentioned in [19], let us consider a system that handles a video stream between the producer and the consumer connected by a communication network such as wireless and mobile one. The consumer processes the video frames with the algorithms such as object detection by a neural network. The application demands higher resolution for improving the accuracy in the consumer side. However, in the case when the bandwidth is not stable, controlling the output data rate to the network, the producer changes the resolution and the frame rate to avoid frame drops. One of the traditional methods for the flow controls uses a threshold of FIFO buffer of the video frames in the producer side. In addition, we can find that one of the typical controls for the data rate is a up/down sampling of video frames [20]. It is controlled by a sampling parameter. Combining the sampling rate

with the FIFO's threshold, the system can control the flow in the network. Here, when we apply the stream-based lossless compression for the video stream, we will have a problem how to convey the sampling rate from the producer to the consumer in the compressed data stream. Although we can prepare a separate channel that only transfers the sampling parameter, it is hard to synchronize the timings between when the rate is changed and when the border of the video stream comes in the different sampling rates. Furthermore, when we use a single channel for transferring the rate parameter and the video stream, we will need the framing or the packeting methods as shown in Figure 4. However, if the exception symbol is available, we are able to control the timing by inserting an exception symbol between the parameter data and the subsequent video stream. Thus, the rate control will become simple due to the exception symbol.

As we explained in this section, implementations of the application examples above will become simple. Because we can eliminate the complex controls, the performance of the systems will also be improved. Now, let us propose the novel methods and the implementations for the exception symbol.

3.2. Method for Exception Control

The exception symbol is an irregular event that is treated as other data than compressed data. To generate this irregular event, we can find a hint in the look-up table operation. In our mechanism, the compressor inserts the exception symbol after flushing compressing data in the processing pipeline. Then, when the decompressor receives the exception symbol, it flushes all the data previously received and it can process the exception symbol during the table operations.

In order to implement the mechanism above, we focus on the search operation in the look-up table. The decompressor is able to cause an exception during search operation because the compressed data received has not been decompressed yet. During the search operations, we propose a method that causes an inconsistent search and finally causes an exception. This inconsistent situation is categorized in two methods as shown in Figure 5. In Figure 5a, the compressor outputs an exception symbol as a compressed data, and then the decompressor receives it. The decompressor recognizes it as an index of the look-up table. However, the valid bit of the table entry pointed by the index is false. In this case, the entry does not have any registered symbol. This is inconsistent in the decompression process and detected as an exception. We call this method *Transfer Index*. Figure 5b shows another method. When the compressor outputs an original data to be registered in the decompressor side, the decompressor receives it and tries to register the original data into the look-up table. However, the original data is already registered and found in the table. This is also inconsistent during the table operation in the decompression. This also causes an exception. We call this method *Transfer Original*.

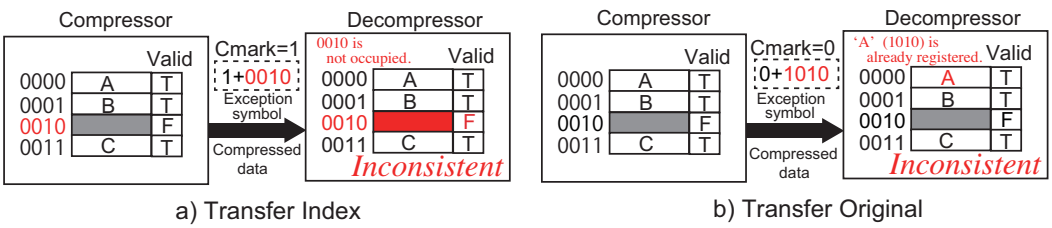

Figure 5. Inconsistent situation in look-up tabe to detect an exception symbol.

These two inconsistent situations also have two categories according to the management of the entries in the look-up table. One is a method that the entry related to an exception is fixed in an index. Indeed, we can implement it by reserving an entry with a symbol that corresponds to a bit pattern of the demanded exception symbol. This method needs to reserve an entry on both cases of the Transfer Index and the Transfer Original. We call this method *Reserve Entry*. The decompressor also reserves the same entry as the

compressor. Another method does not reserve any entry in the look-up table. we call this method *Free Entry*. The compressor picks up a symbol from an occupied entry in the case of Transfer Original, and outputs it as an original data to the decompressor.

As follows the categorization of the methods above, we can have available combinations: Reserve Entry, Transfer Original (RETO), Reserve Entry, Transfer Index (RETI), Free Entry, Transfer Original (FETO) and Free Entry, Transfer Index (FETI). Let us explain these methods in the following sections.

3.3. RETO: Reserve Entry, Transfer Original

RETO reserves one or more entries in the look-up table and initially registers any symbols in the entries on both the compressor and the decompressor. The registered symbols are the exception symbols. The compressor outputs the symbol as an original data. When the decompressor receives the original data, it tries to register the symbol to the look-up table. However, the decompressor detects an inconsistent that the symbol is already registered in the table. Using this inconsistent event, the decompressor knows the original data is an exception symbol.

RETO is similar to the method in LZW because the exception symbol is reserved in the look-up table and it is never invalidated. However, RETO differs in the aspect that the reserved entry joins in the compression operation. In the LZW method, there is no way to detect if the received symbol is a compressed data or not because the compressed data consists of the table index that equals to the registered symbol in the entry. Therefore, the compressor of LZW inevitably uses a reserved entry as an exception symbol and directly outputs the symbol. Thus, the entry is not used during compression process. On the other hand, RETO compresses the symbol reserved in an entry in the table.

RETO has a drawback in the complexity of the algorithm. When the decompressor receives any original data, it needs to search the symbol in the look-up table and checks if the symbol is registered in the table or not. If the symbol is found in the table, it is detected as an exception symbol. If not, it is an original data to be newly registered. Thus, RETO needs to be implemented either by software or in hardware with a search mechanism of the table. Therefore, the complexity of RETO is $O(E)$ where E is the number of entries in look-up table.

In the aspect of compression performance, RETO is able to use the reserved entry for compression. When a symbol inputted to the compressor is corresponding to the symbol registered in the reserved entry in the look-up table, the compressor is able to compress the symbol by converting it to the index of the entry. For example, RETO can prepare the reserved entry by a symbol that is frequently appeared in the inputted data stream to the compressor. This might contribute to achieve a better compression ratio than the case without the reserved entry.

Figure 6 shows examples of RETO applying to LCA-DLT and ASE coding. In the case of LCA-DLT as depicted in Figure 6a, an entry in the look-up table is reserved in both the compressor and the decompressor. The index of the entry must be the same among those. The frequent counter in the entry is not decremented during the table operations and the valid bit is set always to be true. This prevents the entry from removing by the table operations. On the other hand, in the case of ASE coding as shown in Figure 6b, the entry in the lowest index of the look-up table is reserved. Moreover, the registration of a new symbol to the table is preformed from the second lowest index. Here, note that the number of occupied entries k in the entropy calculation is always more than 0 due to the reserved entry. Additionally, the entropy culling targets to invalidate the entries of the indices larger than 0.

As we explained above, RETO reserves an entry in the look-up table and treats an original data as an exception symbol. It is able to use the entry during the compression operation. However, the content of the entry is always fixed. Therefore, the compression performance of RETO is affected by the reserved entry.

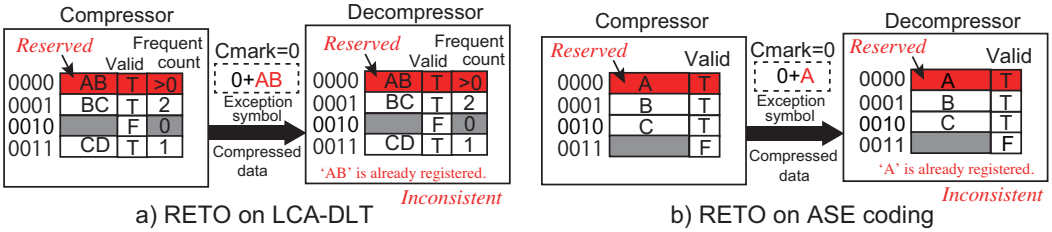

Figure 6. RETO examples applying to (**a**) LCA-DLT and (**b**) ASE coding.

3.4. RETI: Reserve Entry, Transfer Index

In the case of RETI, the compressor outputs a compressed data as an exception symbol. The compressed data is an index of the look-up table. RETI reserves an entry of the look-up table that is not involved in the registration for a new symbol. The entry must be always marked as unoccupied. When the decompressor receives the compressed data, it uses the compressed data as an index of the table and then detects an inconsistency because the entry is reserved as unoccupied. Due to the inconsistency, an exception has occurred in the decompressor.

In RETI, the reserved entry is not used for the compression because the reserved entry in the look-up table must be always unoccupied. This degrades the compression ratio because the number of entries joined in the compression in the look-up table is always $E - k$ where E is the total number of entries physically allocated and k is the number of exception symbols. Thus, RETI is not an optimal method if application expects good compression ratio. However, as we discussed the event-driven exception above, we can set $k = 1$. The degradation of the compression ratio cannot be large.

When we implement RETI in LCA-DLT, we need a mechanism to control the reserved entry to be unoccupied. As shown in Figure 7a, the valid bit of the reserved symbol is fixed to be false permanently. The index of the reserved entry must be equal among the compressor and the decompressor. Here, using the characteristics that the index of the reserved entry should be known by the compressor and the decompressor, the decompressor can detect the exception symbol without touching the look-up table by comparing the reserved index with the known exception symbol. To avoid this irregular mode in the table operations, the straightforward implementation is to associate the index of the table from the exception symbol. Then the inconsistency is detected as the exception. On the other hand, in the case of ASE coding as illustrated in Figure 7b, the entry in the lowest index is reserved as unoccupied. The entry is ignored during the table operations. The entropy culling can include the entry. However, there is no effect to the compression mechanism even if it includes the entry because the reserved entry is always unoccupied. Note that the entropy calculation uses $k \geq 0$ for the number of occupied entries in the table because the number of bits shrunk in the compressor must guarantee the case when the table is empty. However, after an entry is occupied, k becomes 2. This results the entropy calculation is $m \geq 1$. In the other words, the bit length of the compressed data never becomes zero due to the reserved entry. Therefore, we can expect that RETI degrades the compression ratio.

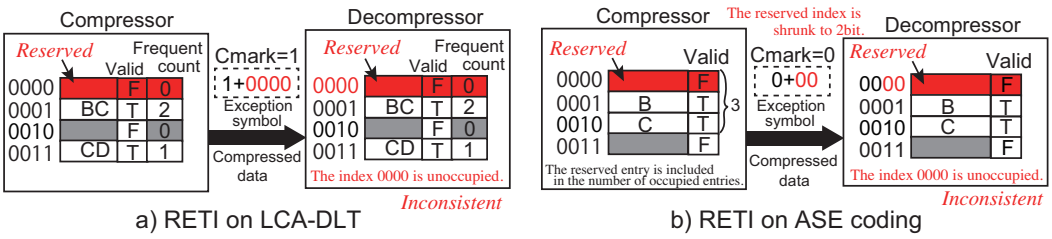

Figure 7. RETI examples applying to (**a**) LCA-DLT and (**b**) ASE coding.

As we can see in this section, RETI implements the exception symbol without any search operation during table operations. However it inevitably degrades the performance because the reserved entry is not used for the compression.

3.5. FETO: Free Entry, Transfer Original

We explained the methods of Reserve Entry above. From this section, we discuss the ones of Free Entry that any entry in the look-up table is not reserved.

In FETO, the compressor outputs an original symbol as an exception symbol as well as RETO. Then, the decompressor detects that the symbol is registered in the look-up table. This is inconsistent in the decompression operation. Thus, an exception has occurred. The original symbol outputted from the compressor is selected one of the registered symbols in the occupied entries in the look-up table. This mechanism has a drawback when the look-up table is empty. If no occupied entry exists in the table, FETO cannot pick up any symbol for the exception. Therefore, one or more entries in the table must be occupied. However, FETO does not affect the compression performance because the compressor is able to use all entries in the look-up table during the compression operations.

Let us see the implementation of FETO in LCA-DLT and ASE coding. First, in the implementation in LCA-DLT as shown in Figure 8a, the compressor picks up an entry from the lookup table that the valid bit is set. Then, it outputs the symbol as the compressed data combining with Cmark = 0. When the decompressor receives it, it tries to register to the look-up table because the compressed data is the original symbol. However, the decompressor scans the table with comparing the received symbol with contents of all the entries. If an entry that matches to the received symbol is found, it is detected as an exception symbol because the situation is inconsistent. Thus, the exception occurs. If not found, it is registered to the table as a new symbol. On the other hand, in the case of ASE coding as shown in Figure 8b, the compressor can choose the lowest entry in the look-up table as an exception symbol because the table content is maintained equally among the compressor and the decompressor. The symbol is outputted from the compressor as well as LCA-DLT. Then, the decompressor just compares the lowest entry in the look-up table. Here, note that we do not need to implement the search mechanism to check if the received symbol is already registered in the look-up table or not. If the comparison fails, the received symbol is pushed to the table. Here, the table operation includes all entries in the table because there is no reserved symbol. Be aware that FETO does not work when the look-up table is empty in both cases of LCA-DLT and ASE coding. This is a fatal drawback of the mechanism. However, by combining FETI explained in the next section, we can implement a perfect mechanism of the exception symbol.

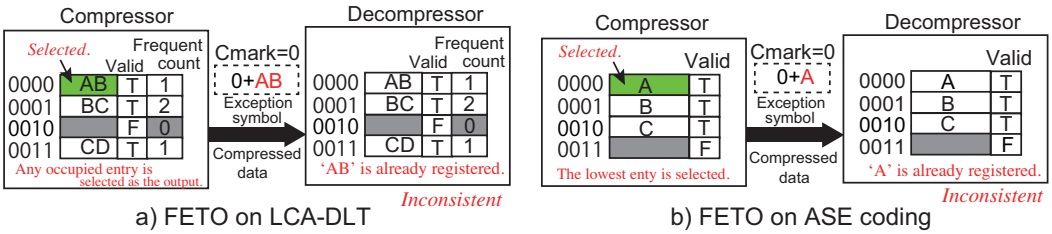

Figure 8. FETO examples applying to (**a**) LCA-DLT and (**b**) ASE coding.

As we discussed above, FETO needs a symbol search operation before the table operation depending on the management algorithm of the table because it does not define any reserved symbol in the look-up table. For example, LCA-DLT needs the search mechanism. However, ASE coding does not need the search mechanism because it can identify the exception symbol by just picking up the lowest entry of the table. Regarding the compression performance, we can expect that FETO does not have any effect to the performance according to the exception symbol because it is able to use all allocated entries in the look-up table. However, FETO does not work when the table is empty.

3.6. FETI: Free Entry, Transfer Index

In FETI, the compressor picks up an index of an unoccupied entry in the table as the exception symbol and outputs it as a compressed data. When the decompressor receives the compressed data, it uses the data as an index of the look-up table to pick up the associated original symbol. However, the entry pointed by the index is unoccupied. This is an inconsistent in the table operation. Thus, the exception has occurred in the decompressor. This mechanism needs to guarantee that at least an unoccupied entry exists in the look-up table. Therefore, it is obviously expected that this mechanism does not work if the table is full. However, this mechanism does not affect the compression performance because it does not need to reserve any entry in the table.

The implementations of FETI differ depending on the algorithms, especially depending on the table management. For example, in the case of LCA-DLT as shown in Figure 9a, the compressor needs to search an unoccupied entry in the look-up table. Here, the compressor can choose any unoccupied entry that the valid bit is reset, and outputs the index as a compressed data with Cmark = 1. The decompressor receives the compressed data and tries to pick up the original symbol from the look-up table. However, the entry pointed by the index is not occupied. This is inconsistent and causes an exception. Note that in LCA-DLT, we cannot apply FETI when the table is full. On the other hand, in ASE coding, we need to implement FETI with a care for the table management operation. FETI is not available when the number of occupied entries equals 2^i where $i \geq 0$. For example, when the number of occupied entries in the look-up table is 16 (the total number of entries initially allocated in the table is more than that), if the compressor generates an exception symbol, it can choose the index of the 17th entry in the table. However, the number of occupied entries does not change. Therefore, the entropy calculation returns $4 = \log_2 16$ and the compressor shrinks the exception symbol to 4bits. However, the number of bits to express the exception symbol (i.e., 16) is five. This lacks a bit of the exception symbol. Thus, FETI does not work in some conditions depending on the number of occupied entries. To avoid this situation, the compressor should activate FETI only when the look-up table is empty. Under the condition when the number of occupied entries in the look-up table is zero, the compressor can simply select the lowest index (i.e., 0) as the exception symbol and outputs it. Due to the entropy calculation $m = ceil(\log_2 0) = 0$ when the table is empty, the compressed data is always a single bit consisted of Cmark = 1 only. Here, the compressor can ignore the index selection and just can output Cmark bit. When the decompressor receives the compressed data, it detects the Cmark and extracts the number of bits due to the entropy calculation as well as the one in the compressor. Then, the table index becomes zero. The lowest entry in the table is selected. However, it is not occupied. This is inconsistent in the decompression algorithm and thus, an exception has occurred. Now, let us see a case when several entries are occupied in the look-up table as shown in Figure 9b). In this case, the entropy calculation results $m = ceil(\log_2 3) = 2$. The index 0011 is selected as an exception symbol because it is not occupied and is placed in the lowest. It is shrunk to 11 and Cmark = 1 is combined. In the decompressor side, the compressed symbol is recognized and the index 0011 is referred. However, it is not occupied. Thus, the exception occurs.

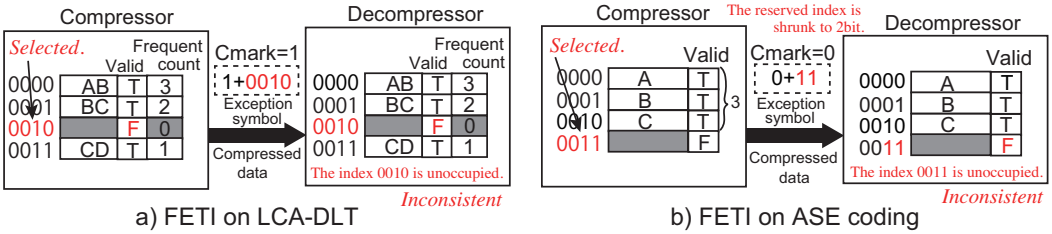

Figure 9. FETI examples applying to (**a**) LCA-DLT and (**b**) ASE coding.

As mentioned above, FETI does not affect the compression performance because any entry in the look-up table is not reserved. However, this method is not available if the look-up table is full.

3.7. Implementation Consideration

Here, let us consider optimal combinations of four methods explained above.

We discussed above that RETI and RETO can degrade the compression performance due to the reserved entry in the look-up table. Therefore, it would not be suitable when application needs good compression ratio. However, in the aspect of implementation complexity, RETI is simpler than RETO because it does not need the symbol search mechanism in the decompressor side to find the corresponding symbol to the exception symbol.

On the other hand, the combination of FETO and FETI will implement the exception symbol without performance degradation. FETO is not available when the look-up table is empty. On the other hand, FETI is not available when the table is full. Therefore, the combination of those two methods will achieve the best solution, i.e., FETI is activated when the table is empty and FETO is done when the table is not empty. In this combination, the compressor does not need to reserve any entry in the table and never degrades the compression performance because the compressor can use all entries in the table.

We explained the exception handling method in the digram coding with the look-up table that the number of entries is fixed. The major method for the exception is to make an inconsistent state in the look-up table operation. We categorized the mechanism into four methods. According to the discussion focusing on the implementation difficulty and the compression performance, the optimal implementation of the exception symbol is FETI+FETO because the compression performance never degrades. However, if application desires the simpler implementation, although the compression performance degrades, RETI is the best solution. Here, we have a question about how much the performance degradation occurs by RETI and RETO. Let us observe the performance from experimental evaluations in the next section.

4. Experimental Evaluations

We will discuss the compression performance applying RETO, RETI, FETO and FETI to LCA-DLT and ASE coding. We will compare the performance of three implementations among RETO, RETI, FETI+FETO.

During the evaluations, we employ benchmark data available from [21–24]. From the benchmarks, we use two ASCII text data and two image data. The text data potentially includes some rules based on ASCII code. This provides frequency that makes the entropy low. On the other hand, the image data is randomly generated depending on the image sensor or the graphics rendering algorithm. This provides high entropy than the text data. In the evaluations, we just compare the performance without exchanging any exception symbol between the compressor and the decompressor. Therefore, we will observe the compression performance by just applying the benchmark data to the compressor with the mechanisms of RETO, RETI and FETI + FETO.

The gene DNA sequences and the English text are picked up from [21]. The contents of the files are organized with ASCII text data sequences. We use the first 10Mbyte of the downloadable file from the site. The *Beauty* is picked up from the second website. It is provided by a file with a video frame sequence in YUV420 format in 8 bit depth. We use the 100th frame of the sequence in the video file. The size of the frame data is 12 Mbyte. The *Sintel* is picked up from [24]. It is a video file of a computer graphics amination which frame size is 10 MByte. We use the 1000th frame of the Sintel formatted in YUV420 of 8 bit depth by converting from the TIFF file. Here, let us explain the reason we chose these data sequences as the benchmark. The gene DNA sequences consist of only four characters (A, T, G, C) of eight bit wide. The English text is an example of a more complex data patterns of alphabets and some other symbols with eight bit wide. The evaluation using this will show a compression performance of a data sequence with higher entropy

than the gene DNA sequence. The Beauty is an example of a natural image data which resolution is 3840 × 2160. Note that in the YUV420 format, Y:U:V is 4:1:1. It stores 8 bit Y element for every pixel. It also stores U and V elements that are also 8 bits respectively derived from 2 × 2 pixel block. The evaluation with this data sequence will show effects of the algorithms in the case of high entropy data stream. On the other hand, the Sintel is an example of a data sequence based on an artificial image which resolution is 4096 × 1744 formatted in YUV420. The data size is 10 MB. This includes frequent color patterns due to the creation algorithm of the computer graphics. Therefore, the evaluation using this image data will show effects when we apply the algorithms to such data sequence with lower data entropy than the natural image.

In the experiments, we apply the methods of the exception symbol to LCA-DLT under the following configurations: the initially allocated entries in the look-up table varies from 16 to 256. The symbol width is 8 bit. We also apply the methods to ASE coding under the following configurations: the initially allocated entries in the look-up table varies from 4 to 64. The symbol width is 8bit. The entropy culling is set to 2. The methods for the exception symbol are implemented by applying the examples explained in the previous Sections 3.3–3.6 depicted in Figures 6–9. FETO+FETI is the combination discussed in Section 3.7 that FETO is applied when the look-up table is empty and otherwise FETI is applied. Note that the performance of FETI+FETO equals to the one without the exception symbol. We also perform an evaluation of RETO with the most frequent value (denoted as RETO (MFV)) that sets the most frequent symbol in the benchmark data to the reserved entry. In the case of LCA-DLT, we pick up the most frequent pair of contiguous symbols from the benchmark data, which is 16 bits. The percentages of the values are 4.5%, 1.4%, 0.16% and 2.4% in gene DNA sequences, English text, Beauty and Sintel respectively. On the other hand, in the case of ASE coding, we use the most frequent 8bit symbol from the benchmark data. The percentages of the values are 28.6%, 17.4%, 2.16% and 5.18% in gene DNA sequences, English text, Beauty and Sintel respectively. The compression ratio is the metric of the performance comparison, which is calculated by (*compressed data size/original data size*) ×100 (%). The smaller the ratio is, the better the compression performance is achieved. As reference performances of the compression ratios by ZIP (the default setting of Info-ZIP 3.0), we confirmed that the ratios of gene DNA sequences, English text, Beauty and Sintel resulted 28.26%, 37.92%, 70.60% and 14.66%.

Figure 10 shows the performance comparisons among different configurations in LCA-DLT with the methods of the exception symbol. All performances show almost the same compression ratios in any configurations. We confirmed that the mechanism of the exception symbol did not have overhead in the compression performance. RETI and RETO reserves an entry in the look-up table. Therefore, we expected that the effect of the reserved entry appears in the compression ratio when the number of entries is small. However, it does not affect much to the compression ratio because the compression mechanism recovers the disadvantage dynamically according to the data entropy. On the other hand, when the number of entries in the look-up table is small, potentially the compression ratio becomes worse such as the one more than 100%. Even when RETO (MFV) sets the most frequent symbol pair in the reserved entry, the advantage of the reserved symbol is not observed at all because the compressor does not increase the hit ratio against the look-up table due to the reserved symbol. This is caused by the mechanism that the frequent counter is incremented at hit operations. Therefore, the frequent symbol pairs can stay in the table for a long time. This allows the compressor to keep the frequent symbols in the table, and contributes to compress the symbols. Finally, the case without the reserved entry even results the similar performance to the case with the one. Thus, we conclude that the proposed methods of the exception symbol do not affect the compression ratio in LCA-LDT.

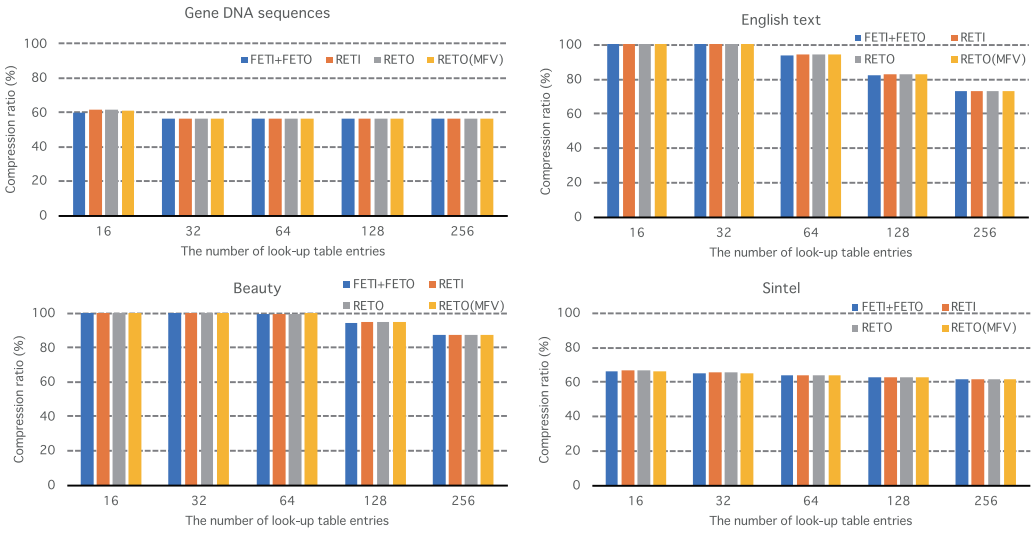

Figure 10. Comparison of compression performance among RETO, RETI, FETO+FETI (without the exception symbol) and RETO(MFV) (maintaining the most frequent value in the reserved entry) applying on LCA-DLT.

Figure 11 also shows the performance comparisons among different configurations in ASE coding with the methods of the exception symbol. We should focus on the compression performance of RETI and RETO. The method of the reserved entry in the table has an impact on making the bit length of the compressed data longer because the lowest entry is always occupied. This makes m from the entropy calculation $m \geq 1$. On the other hand, in the case of FETI + FETO, the entropy calculation returns $m \geq 0$, i.e., the method to reserve an entry inevitably needs to increase a bit in the compressed data. Therefore, the main reason for the performance difference between RETI/RETO and FETI + FETO is the effect from the additional bit according to the reserved entry. However, we found that RETO can cancel the performance decrease by using the reserved entry with setting the most frequent symbol to the entry. The result of RETO(MFV) shows almost equivalent compression performance to the one of FETI + FETO. In the case of ASE coding, RETO has advantage in the simple implementation. Therefore, if the most frequent symbol is known originally, this method helps in both aspects of implementation and compression performance.

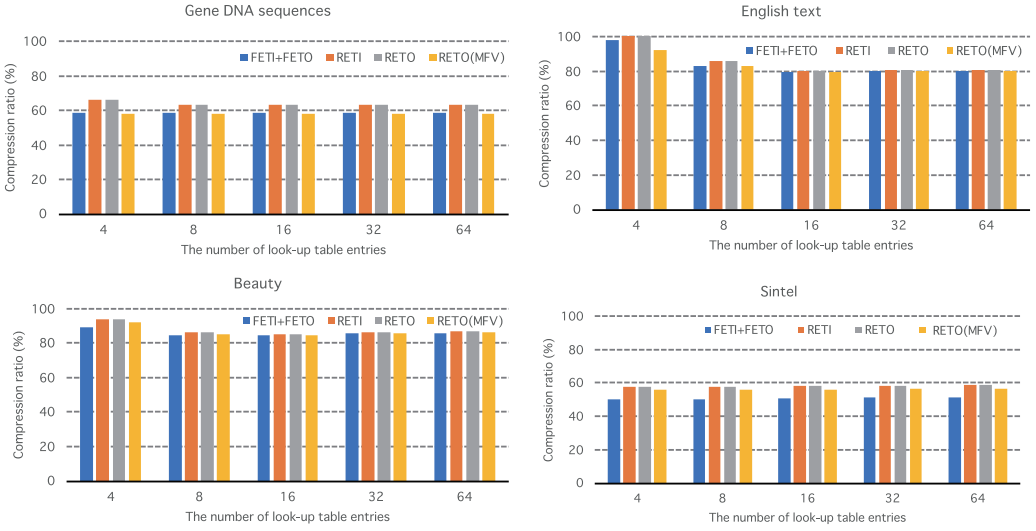

Figure 11. Comparison of compression performance among RETO, RETI, FETO + FETI (without the exception symbol) and RETO(MFV) (maintaining the most frequent value in the reserved entry) applying on ASE coding.

As we discussed in the previous section, the graphs from the results of LCA-DLT and ASE coding shows that FETI+FETO is the best solution to handle the exception symbol because it does not affect the compression performance at all and also exploits the potential performance of the compressor.

According to the performance evaluations above, the exception methods proposed in this paper implement appropriate exception handling on the digram coding with a fix-sized look-up table. In particular, we designed the methods for the stream-based lossless compression that needs to compress/decompress a contiguous data stream without stalling in a small delay. Thus, we conclude that we implemented effective exception handling methods that do not affect the compression performance.

5. Conclusions

This paper proposes exception handling methods for the digram coding that the number of entries in the look-up table is fixed. We proposed the method to handle the exception symbol in the lossless data compression. We also classified the method into four categories called RETO, RETI, FETO, and FETI according to the detection mechanism for the exception during the management operations for the look-up table. We explained the implementation examples on the stream-based lossless compression algorithms, LCA-DLT and ASE coding, and also discussed the effects for the compression performance. Regarding the effects for the performance, we performed the experimental evaluations applying the methods for the exception symbol on the stream-based lossless data compression algorithms. According to the comparisons of the compression ratios from the experiments, we confirmed that the reserved methods have slight performance degradation. However, FETI + FETO implements the stream-based data compression without any overhead in the compression performance. Thus, we proposed an exception handling method in the digram coding with a fixed-size look-up table by eliminating overhead in the compression performance. For the future plans, we are going to apply the proposed methods to applications that need to communicate combined data stream via slow communication devices such as Bluetooth. Then, we will validate the proposed methods in realistic applications.

Author Contributions: Conceptualization, S.Y.; methodology, S.Y. and K.M.; software, S.Y. and S.K.; validation, S.Y., K.M. and S.K.; formal analysis, S.Y.; investigation, S.Y.; resources, S.Y.; data curation, S.Y.; writing—original draft preparation, S.Y.; writing—review and editing, S.Y.; visualization, S.Y.; supervision, S.Y.; project administration, S.Y.; funding acquisition, S.Y. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by JSPS KAKENHI Grant Number 20H04152, JST CREST Grant Number JPMJCR1402 and JST PRESTO Grant Number JPMJPR203A.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Marumo, K.; Yamagiwa, S.; Morita, R.; Sakamoto, H. Lazy Management for Frequency Table on Hardware-Based Stream Lossless Data Compression. *Information* **2016**, *7*, 63. [CrossRef]
2. Yamagiwa, S.; Hayakawa, E.; Marumo, K. Stream-Based Lossless Data Compression Applying Adaptive Entropy Coding for Hardware-Based Implementation. *Algorithms* **2016**, *13*, 159. [CrossRef]
3. Mohammadi, M.; Al-Fuqaha, A.; Sorour, S.; Guizani, M. Deep Learning for IoT Big Data and Streaming Analytics: A Survey. *IEEE Commun. Surv. Tutor.* **2018**, *20*, 2923–2960. [CrossRef]
4. Howard, P.G.; Vitter, J.S. A universal algorithm for sequential data compression. *Inf. Process. Manag.* **1992**, *28*, 749–763. [CrossRef]
5. Langdon, G.G. An Introduction to Arithmetic Coding. *IBM J. Res. Dev.* **1984**, *28*, 135–149. [CrossRef]
6. Martin, G.N.N. Range encoding: An algorithm for removing redundancy from a digitised message. In Proceedings of the Video and Data Recording Conference, Southampton, UK, 24–27 July 1979.
7. Huffman, D.A. A Method for the Construction of Minimum-Redundancy Codes. *Proc. IRE* **1952**, *40*, 1098–1101. [CrossRef]
8. Vitter, J.S. Design and Analysis of Dynamic Huffman Codes. *J. ACM* **1987**, *34*, 825–845. [CrossRef]
9. Ziv, J.; Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **1977**, *23*, 337–343. [CrossRef]
10. Ziv, J.; Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* **1978**, *24*, 530–536. [CrossRef]

11. LZ4. Available online: https://lz4.github.io/lz4/ (accessed on 5 December 2020).
12. Google. Available online: https://github.com/google/snappy (accessed on 5 December 2020).
13. Deutsch, P. *RFC 1951 DEFLATE Compressed Data Format Specification Version 1.3*; Aladdin Enterprises: Jaw, SK, Canada, 1996.
14. Welch, T. A Technique for High-Performance Data Compression. *Computer* **1984**, *17*, 8–19. [CrossRef]
15. Yamagiwa, S.; Sakamoto, H. A reconfigurable stream compression hardware based on static symbol-lookup table. In Proceedings of the 2013 IEEE International Conference on Big Data, Santa Clara, CA, USA, 6–9 October 2013; pp. 86–93.
16. Maruyama, S.; Sakamoto, H.; Takeda, M. An Online Algorithm for Lightweight Grammar-Based Compression. *Algorithms* **2012**, *5*, 214–235. [CrossRef]
17. Yamagiwa, S.; Marumo, K.; Sakamoto, H. Stream-based Lossless Data Compression Hardware using Adaptive Frequency Table Management. In Proceedings of the Very Large Data Bases/BPOE 2015, Kohala, HI, USA, 31 August–4 September 2015.
18. Marumo, K.; Yamagiwa, S. Time-Sharing Multithreading on Stream-Based Lossless Data Compression. In Proceedings of the 2017 Fifth International Symposium on Computing and Networking (CANDAR), Aomori, Japan, 19–22 November 2017; pp. 305–310.
19. Kua, J.; Nguyen, S.H.; Armitage, G.; Branch, P. Using Active Queue Management to Assist IoT Application Flows in Home Broadband Networks. *IEEE Internet Things J.* **2017**, *4*, 1399–1407. [CrossRef]
20. Tang, T.; Yang, J.; Du, B.; Tang, L. Down-Sampling Based Rate Control for Mobile Screen Video Coding. *IEEE Access* **2019**, *7*, 139560–139570. [CrossRef]
21. Compressed Indexes and Their Testbeds. Available online: http://pizzachili.dcc.uchile.cl/ (accessed on 5 December 2020).
22. Ultra Video Group. Available online: http://ultravideo.cs.tut.fi/ (accessed on 5 December 2020).
23. Mercat, A.; Viitanen, M.; Vanne, J. UVG Dataset: 50/120fps 4K Sequences for Video Codec Analysis and Development. In Proceedings of the 11th ACM Multimedia Systems Conference, MMSys '20, Istanbul, Turkey, 8–11 June 2020; ACM: New York, NY, USA, 2020; pp. 297–302. [CrossRef]
24. Xiph.org Test Media. Available online: https://media.xiph.org/ (accessed on 5 December 2020).