Chapter 16

Stream-Based Lossless Data Compression

Shinichi Yamagiwa

Abstract In this chapter, we introduce aspects of applying data-compression techniques. First, we study the background of recent communication data paths. The focus of this chapter is a fast lossless data-compression mechanism that handles data streams completely. A data stream comprises continuous data with no termination of the massive data generated by sources such as movies and sensors. In this chapter, we introduce LCA-SLT and LCA-DLT, which accept the data streams, as well as several implementations of these stream-based compression techniques. We also show optimization techniques for optimal implementation in hardware.

16.1 Introduction to Stream-Based Data Compression

Rapid communication data paths are demanded in computer systems to improve performance, and the fastest data paths have recently reached the order of tens of gigahertz as implemented by optical fiber. One solution to achieving rapid communication data paths is to have parallelized paths in multiple connections, but technological trials have offered no clear solutions because of electrical and physical limitations such as crosstalks and refractions. To overcome the problems associated with high-speed communication, this chapter focuses on data compression on the data path. There are two ways in which this can be implemented. One is software-based compression, which is typically implemented on the lower layer of the communication data path, such as the device-driver level of Ethernet [18]. The other way is hardware-based implementation, which must provide low latency and stream-based compression and decompression.

Well-known algorithms such as Huffman encoding [17] and Lempel-Ziv-Welch (LZW) compression [21, 22] perform data encoding by creating a symbol lookup table (LUT), in which frequent data patterns are replaced by compressed symbols

S. Yamagiwa (✉)

Faculty of Engineering, Information and Systems, University of Tsukuba, 1-1-1 Tennodai, Tsukuba, Ibaraki 305-8573, Japan

e-mail: yamagiwa@cs.tsukuba.ac.jp

JST, PRESTO, 4-1-8 Honcho, Kawaguchi, Saitama 332-0012, Japan

© The Author(s) 2022

N. Katoh et al. (eds.), *Sublinear Computation Paradigm*,
https://doi.org/10.1007/978-981-16-4095-7_16

in the table. However, hardware implementation presents the following difficulties: (1) the processing time is unpredictable because the data length is not deterministic, (2) maximal memory must be prepared because the lengths of the data patterns are not deterministic, and (3) blocking decompression is performed. Here, we focus on a stream-based lossless data-compression mechanism that overcomes these problems. The key technology is a histogram mechanism that caches the compressed data. The decompressor must manage the same table contents as the compressor side and reproduce the original data from the table. In this chapter, we introduce challenges to implementing stream-based lossless compression based on hardware. The ultimate goal is to implement compact and fast data-compression hardware without blocking the compression operations upon accepting continuous data streams. We begin by focusing on a technique with a static LUT, called LCA-SLT, and then we show one with a dynamic table, called LCA-DLT. We also describe performance optimizations for LCA-DLT.

16.2 Stream-Based Lossless Data Compression with Static Look-Up Table

16.2.1 *Design of LCA-SLT*

We begin by focusing on a compression algorithm called online LCA (Lowest Common Ancestor) [12], which converts a symbol pair to an unused symbol with the LUT of symbol pairs managed as shown in Fig. 16.1, which shows an example of compressing the sequence ABCDFFBC to the symbol Z. Online LCA addresses the problems caused by conventional dynamic LUT management and provides a fixed time complexity due to the two-symbol matching. During the decompression, online LCA invokes the opposite mappings by repeating conversions from one symbol to two according to the table starting from the deepest compression step.

Applying the concept of online LCA, we show here the mechanism of LCA-SLT (LCA Static Look-up Table) [20], which prepares statically allocated LUTs that are used for converting symbol pairs. The compressor encodes inputted symbols using the LUTs, and the decompressor does the opposite. The contents of the tables are stored statically and initially before the compression/decompression. The tables are prepared heuristically in the following steps: (1) a test set of the target data is examined by online LCA, (2) the LUTs are created from all the original symbol pairs and their matching symbols, (3) the entries in the LUTs are sorted in ascending order by frequency, and finally (4) the entries in the top ranks are registered as the table contents. These steps implement the best matching patterns in the original data set as determined by the frequency analysis.

As shown in Fig. 16.2, the compressor and decompressor perform online LCA using the tables created from a set of test data patterns. The modules are connected

Fig. 16.1 LCA example. If the pairs AB and CD can be converted to K and L, respectively, then the original data become KL. If KL can also be converted to O, then the next pair becomes OP

Fig. 16.2 LCA-SLT module comprising a compressor, LUT for compression, decompressor, and LUT for decompression

from the compressed and decompressed data lines one after another and organize a pipeline for recursive compression/decompression operations.

The method with static LUTs has two main advantages. First, the compressed data never include any additional information for table management. Second, the amount of table resources is deterministic. Therefore, LCA-SLT can be implemented on compact hardware and is fast because of its simple compression/decompression operations.

16.2.2 Implementation of LCA-SLT

On hardware, the compressor and decompressor can be implemented using a content-addressable memory (CAM) [8] and a normal memory (MEM), respectively. The CAM is a type of hardware into which a set of data bits is inputted and that outputs a matched address where the data are stored. Figure 16.3 shows the organization of the compression part. As an example, the combination of two symbols becomes 16 bits when the symbol width is 8 bits, and we add another bit per compressed data to mark whether it is compressed, called the *compression mark (CMark) bit*. Figure 16.3 shows a compression pipeline in which four modules are connected. Each module

Fig. 16.4 Performances of LCA-SLT

for a processor interface, it is effective to implement LCA-SLT. On the other hand, the SRL-based implementation shows that the maximal frequency for the input clock will decrease drastically with increasing number of LUT entries. In the FPGA case, we must consider how the number of table entries affects the performance because the limited number of physical wires in the large-scale integration decreases the routing availability when the matching address bits due to the CAM become wide.

Thus, the LCA-SLT implements a compression mechanism with small overhead for data streams. It is reconfigurable depending on the characteristics of the target data, addressing the desired performance depending on the number of compression/decompression modules or the number of bits in a symbol or the available symbol mapping entries in the LUT.

Table 16.1 Compilation reports regarding hardware implementations of LCA-SLT

128 entries	# of slice registers	# of slice LUTs	# of BRAMs	Max freq.
Using SRL-based CAM	141	2224	1	93 MHz
Using BRAM-based CAM	380	738	21	93 MHz
256 entries	# of slice registers	# of slice LUTs	# of BRAMs	Max freq.
Using SRL-based CAM	144	4135	1	75 MHz
Using BRAM-based CAM	638	1546	41	101 MHz
512 entries	# of slice registers	# of slice LUTs	# of BRAMs	Max freq.
Using SRL-based CAM	152	8128	1	51 MHz
Using BRAM-based CAM	1152	2567	81	81 MHz

16.3 Stream-Based Lossless Data Compression with Dynamic Look-Up Table

16.3.1 Design of LCA-DLT

Next, we focus on another algorithm for stream-based data compression with dynamic table management, called LCA-DLT (LCA Dynamic Look-up Table) [19]. It allocates corresponding symbol LUTs for the compressor and the decompressor, respectively. Each table has any number N of entries and the i -th entry E_i includes a pair of the original symbols ($s0_i, s1_i$), a compressed symbol S_i , and a frequent counter $count_i$. The compressor side uses the following rules: (1) reading two symbols ($s0, s1$) from the input data stream and if they match to $s0_i$ and $s1_i$ in a table entry E_i , then after incrementing the $count_i$, it outputs S_i as the compressed data; (2) if the symbols do not match to any entry in the table, it outputs ($s0, s1$) and register an entry ($s0_k, s1_k, S_k, count_k = 1$) where S_k is the index number of the entry; (3) if all entries in the table are used, then decrement all $count_i$ ($0 \leq i < N$) until any count(s) become zero, and then delete the corresponding entries from the table. When compressed data S are transmitted from the compressor, the steps in the decompressor are equivalent to those in the compressor. The symbol matching is performed based on S_k in an entry. If the compressed symbol S_i matches to S_k in a table entry, then ($s0_k, s1_k$) is outputted. If not, then another symbol S' from the compressed data stream and the pair (S, S') is outputted and then the pair is registered in the table.

Fig. 16.5 Compression example for the LCA-DLT

Fig. 16.6 Decompression example for the LCA-DLT

When the table entry is full, the same operations as those of the compressor are performed.

Figures 16.5 and 16.6 show examples of compression and decompression operations, respectively. Here, the input data stream for the compressor is ABABCDACABEFD CAB. First, the compressor reads the first two symbols AB and tries to match that pair in the table (Fig. 16.5a). However, the matching fails, and the compressor registers A and B as the *s0* and *s1* in the table. Here, the compressed symbol is assigned in the entry, which is the index 0 of the table. Thus, a rule $AB \rightarrow 0$ is performed. The *count* is initially set to 1. When the compressor continuously reads a pair of symbols (again AB) and it matches in the table, Fig. 16.5b translates AB to 0. Subsequently the equivalent operations are performed. If the table becomes full (Fig. 16.5c), then the compressor decrements the *count*(s) of all entries until any *counts* become zero. Here, three entries are invalidated from the table in the figure. The compressor will register a new entry to the invalidated entry from the smallest index of the table. Figure 16.5d shows that the compressor added a new entry after the invalidation. Finally, the original input data are compressed to AB0CDAC0EFD C0.

The decompressor reads A first (Fig. 16.6a), but it does not match any compressed symbol in the table (because the table is empty). The decompressor then reads another symbol B and registers AB to a new table entry. The entry saves a rule $AB \rightarrow 0$. Thus, the output becomes AB. The decompressor reads the next symbol 0 (Fig. 16.6b), which matches to the table entry. The decompressor translates it to AB and outputs it again. After the subsequent decompression operations, when the table becomes full, the decompressor decrements the *count*(s) as well as on the compressor side (Fig. 16.6c). The invalidated entries must be equivalent to those on the compressor side. Therefore, the compressed symbols are consistently associated with the original symbols. Finally, the compressed data inputted to the decompressor are associated

and outputted as ABABCDACABEFDCAB, which is the same pattern as the input data on the compressor side.

16.3.2 Implementation of LCA-DLT

Figure 16.7 shows an implementation of the LCA-DLT. The input data are propagated through the latches, and the compressed/decompressed data are processed in a pipeline manner. The LUT in the compressor is organized as shown in Fig. 16.8a. The symbol LUT performs the compressed/decompressed data association. Here, the index becomes the compressed symbol, and the enable signal from the matching part increments the *count*. The full management logic of the LUT activates the invalidate control: it decrements the *count* and resets the valid bits (*v* in the figure) regarding the invalidated entry. The LUT in the decompressor is organized with a RAM and a CAM as shown in Fig. 16.8b. The management part of *count* also performs equivalently to that of the compressor based on a CAM. Besides, the matching part is implemented simply in a RAM. The compressed data generated from the address are inputted to the RAM, and the original uncompressed data pair is associated.

The invalidate operation looks for the minimal *counts* in the table entries by decrementing those counts. During the operation, the stall signal is outputted to stop the compression/decompression data pipeline. Figure 16.9a shows an implementation based on parallel decrement logic, and Fig. 16.9b shows one based on serial decrement logic. These two implementations have a tradeoff between the amount of logics and the compression speed when the table becomes full.

In the LCA-DLT as in the LCA-SLT, the compressor adds the CMark bit that indicates whether or not the symbol is compressed. Moreover, by combining the compressor and decompressor in a module and cascading the modules as shown in Fig. 16.10, we can compress long symbol patterns corresponding to 2, 4, 8, or 16

Fig. 16.7 Overall functional block diagrams of the compressor and decompressor in LCA-DLT. The compressor's LUT receives two input symbols from the latches and outputs the selected signal to the multiplexer for the output data. The decompressor's LUT performs the opposite data translation

Fig. 16.8 Detailed organization of LUTs in LCA-DLT. The table has $2n$ entries when a symbol is n bits. The matching part for s_0 and s_1 must be organized as a content-addressable memory (CAM), which outputs the index (i.e., the address in the CAM) matched to an inputted pair of (s_0, s_1) . The management part for count is also organized by a CAM

Fig. 16.9 Decrementing logic for entry invalidation in LCA-DLT

Fig. 16.10 Cascading modules of LCA-DLT. This example compresses long symbol patterns corresponding to 2, 4, 8, and 16 symbols. If the input data at the first compressor are 8 bits long, then the output compressed data become 12 bits because of the CMark bits

symbols when there are four modules. If the input data at the first compressor are 8 bits long, then the output compressed data become 12 bits after four modules because of the CMark bits.

Fig. 16.11 Compression performances of LCA-DLT

16.3.3 Performance Evaluations

Figure 16.11 shows the compression ratios ($(compressed_data_size \div original_data_size) \times 100$). The numbers of table entries are varied from 16 to 256. Focusing on the performance impact of the number of table entries, the compression ratios are improved linearly except for the gene DNA sequence; because the DNA data have a few patterns, all patterns can be saved in 16 entries. Furthermore, focusing on the impact of the number of modules, the compression ratios degrade in the case of more than two modules. This means that a communication data path using too many compression modules becomes disadvantageous because of the CMark bit added after each module.

Figures 16.12 and 16.13 show the hardware performances of the LCA-DLT. It was implemented with only hundreds of slices and a memory block in the FPGA. The LCA-DLT works at 100 MHz with any number of modules, thereby achieving 800 Mbit/s. The LCA-DLT has large impact on resource usage with respect to the logic but not the memory because the recent FPGA does not have any dedicated hardware

Fig. 16.12 Hardware resources of LCA-DLT. It is compiled with 8-bit data input in the first compressor for the Xilinx Artix7 device (XC7A200T-1FBG676C)

Fig. 16.13 Performance comparison between parallel and serial invalidation mechanisms with two modules

macros for CAMs. It is inevitably implemented by LUT and registers in the FPGA. We also compare the amount of hardware resources among the mechanisms of the parallel and the serial invalidations. The parallel version uses larger hardware resources; regarding the dynamic performance of the LCA-DLT, the parallel version involves very few stalls, but its hardware resources explode. Assuming that the hardware works at 100MHz, the effective bandwidth in the input of the first compressor is about 800 and 340–730 Mbit/s with the parallel and serial invalidations, respectively. The output bandwidth of the second compressor will be reduced to 35–80% of the original data size. This means that the LCA-DLT realizes a communication data path that can send more data even if the speed of the path is slow, and it also contributes largely to realizing a high-speed communication data path while providing flexible adjustment between the hardware resources and the compression performance.

16.4 Optimization Techniques for LCA-DLT

Here we introduce optimization techniques for implementing the LCA-DLT. We consider two available optimization techniques: lazy management and time-sharing multi-threading.

16.4.1 Lazy Management of Look-Up Tables

First, we consider the techniques of dynamic invalidation on LUTs and lazy compression [11] that eliminate stalls during the LUT invalidations.

16.4.1.1 Dynamic Invalidation for Look-Up Table

With the management technique of dynamic invalidation for the symbol LUT, we prepare a *remove pointer* and an *insertion pointer*. Initially, the remove pointer points to any entry of the symbol LUT. The $count_i$ is decremented when the pointer comes to the table index i , and if the $count_i$ becomes zero after the decrement, then the entry is removed from the table. The pointer is moved to the next table index after any table search operation. By contrast, the insertion pointer initially points also to any empty entry in the symbol LUT; if the entry is used, then the pointer moves to an unused entry. Using these two pointers, we can expect that a moderate number of the entries occupied in the symbol LUT can be removed.

Figure 16.14 shows an example of the dynamic invalidation mechanism for compression. We assume that DCAADCBBDB is inputted to the compressor and that the remove pointer starts on the second entry of the table. First, DC does not match any entry in the table (Fig. 16.14a), and the compressor waits for an empty entry to appear. The remove pointer is moved to the next entry and the count value is decremented. In Fig. 16.14b, the count value of the third entry becomes zero, whereupon the entry is removed. The insertion pointer is moved to point to the empty entry. The new entry for DC is registered to where the insertion pointer is pointing. Now, DC is outputted. During these operations, the input and output of the compressor stall. When the input symbol pair matches an entry, it is compressed as shown in Fig. 16.14c, d, the remove pointer is moved, and the count value is decremented. If the entry that matches the input symbol pair corresponds to the one pointed out by the remove pointer, then the count value does not change, as shown in Fig. 16.14e. Finally, after the initially inserted DC is removed because of the count value, the entry is used as a new one. Because it was not found in the table, DB is outputted. Thus, the compressed data stream becomes DC012DB.

Figure 16.15 shows the steps of the decompression mechanism using the dynamic invalidation. The inputted compressed data stream is the one generated by the compression in Fig. 16.14. The insertion and the remove pointers begin from the same

Fig. 16.14 Example of the dynamic invalidation mechanism for compression

Fig. 16.15 Example of the dynamic invalidation mechanism for decompression

Fig. 16.16 Example of the lazy compression on compressor side

Fig. 16.17 Example of the lazy compression on decompressor side

entries initially defined by the compressor. Although the matching target is the compressed data, the steps are equivalent to the ones performed on the compressor side. In Fig. 16.15a, b, the I/O of the decompressor stall. When matching the compressed symbol in an entry, the decompressor outputs the corresponding symbol pair such as in Fig. 16.15c, d. Again, a stall occurs during the invalidation of an entry as shown in Fig. 16.15e, f. Finally, the original data stream is decoded.

16.4.1.2 Lazy Compression

Another optimization technique is the lazy compression. This technique ignores compression using the symbol lookup table when the symbol lookup table is full. This eliminates stalls and continuously outputs the data to the decompressor side.

Figure 16.16 shows a compression example of lazy compression applied to the LCA-DLT with dynamic invalidation. First, DC does not match any entry in the table. Here, the lazy compression just passes through the symbol pair without registering the pair into the table. Therefore, no stall occurs as in Fig. 16.16a, e. When the symbol pair matches an entry, the pair is compressed to the corresponding symbol as shown in Fig. 16.16b, d. If the table contains empty entry(ies) when the inputted symbol pair does not match any entry, then it is registered to the empty entry and is also passed through to the output as in Fig. 16.16c. The output from the compressor becomes DC0DC1DB, which is larger than DC021DB for the case of eager compression.

Figure 16.17 shows the case for the decompressor. First, D is not included in the table, therefore the input is the original data pair because actually the CMark bit is added to the compressed data. The compressor does not register the pair and passes through DC to the output as shown in Fig. 16.17a, e without any stall. If the compressed data are in the table, then the decompressor translates the original symbol

Fig. 16.18 Compression ratios with optimizations. The orange lines show lazy compression against the full search method, and the blue ones show dynamic invalidation against the full search method. The results depicted as lines were from using a compressor with four modules

pair such as in Fig. 16.17b, d. If the symbol is not in the table and there are empty entry(ies), then the inputted symbol pair is registered.

16.4.1.3 Performance Evaluations

Figure 16.18 shows the compression ratios with the above optimizations in the LCA-DLT. The bars show the ratios (i.e., the compressed data size divided by the original data size). We can confirm that the lazy compression effectively eliminates stalls and does not disturb the compression, although it does not compress the inputted data when the data pair does not match entries of the symbol LUT. Overall, both of the proposed mechanisms provide more-effective compression ratios than does the full

Fig. 16.19 Stall cycles and the stall ratios against the total clock cycles in LCA-DLT with optimizations

search method. These mechanisms work well if the randomness of the data is high (i.e., the data entropy is high).

We measured the stall clock cycles to compare the dynamic performance of hardware implementation with that of the proposed techniques. We used a Xilinx Artix-7 FPGA XC7A200T-1FBG676C. The full search method works at 100MHz in this device as described in the previous section. By contrast, the implementation with both proposed mechanisms works at 130 MHz because the implementation was simplified by the lazy management of the symbol LUT.

Figure 16.19 shows the stall cycles as the bars and the stall ratios against the total clock cycles as the lines. The total throughput of the data stream becomes much better than that with the full search method. The degradation of the throughput is 30% with the full search but less than 3% with dynamic invalidation. Regarding lazy compression, the compression delay is the number of clock cycles for the input data stream and is also the number of bytes of the input data (i.e., 10M cycles) because lazy compression never causes stalls.

16.4.2 Time-Sharing Multithreading on Compression

16.4.2.1 Design and Implementation of Time-Sharing Multithreading

The time-sharing multi-threading [10] allows the compressor and decompressor to accept multiple different data streams by dividing the dictionary updating operations

Fig. 16.20 Example structure of time-sharing multi-threading (TSM)

among the various input streams. When N data streams are inputted to the compressor/decompressor, the dictionary updating for each data stream allows $N+1$ clock cycles to be inserted to solve the updating problem. For example, Fig. 16.20, shows a structure with two compressors that share the pipeline stages for the dictionary updating operations while accepting two different data streams. This mechanism does not cause any stalls during the input data streams, therefore the bandwidth of a data stream of a whole compressor/decompressor module degrades to $1/N$. However, the clock frequency is expected to increase.

In implementing the compression mechanism, the following operations are assigned to stages of the encoder pipeline for the compressor hardware. The pre-process operation is performed to prepare the subsequent table matching operation, after which the table search operation is performed. The symbol registration operation to the LUT performs registration of symbols, and finally the symbolizing/lookup operations are performed against the LUT. For decompression, the operations are performed in the opposite way to symbolize the compressed data to an original data pair.

Next we discuss an implementation example of time-sharing multi-threading in the LCA-DLT. Assume that there are two input data streams for the compressor/decompressor, and the pipeline of the compressor is organized as shown in Fig. 16.20. The compression in both data streams takes eight cycles to process a data pair, as does the decompression. The compression pipeline consists of the search stage and the registration stage. The search stage compares the contents of the LUT with the incoming data and then creates a match flag list, and the registration stage updates the corresponding entry in the table according to the match flag list. The decompression pipeline consists of the same stages but is organized in the opposite direction.

16.4.2.2 Performance Evaluations

Here we discuss the performance effect of time-sharing multi-threading. The example structure with two data streams per module explained above is implemented on a

Table 16.2 Performance comparisons of the time-sharing multi-threading (TSM)

Xilinx XCKU025	Frequency (MHz)	Combinational logic	Registers	RAM bits
Compressor with TSM	342	7,347	9,909	5,120
Compressor without TSM	277	9,475	7,503	0
Decompressor with TSM	353	2,145	868	17,408
Decompressor without TSM	328	6,063	2,642	4,096

Xilinx Kintex UltraScale FPGA XCKU025-FFVA1156-1-C, and Table 16.2 shows the comparisons. Compared with the clock frequency without time-sharing multi-threading, that with the optimization increases by a factor of approximately 1.23 for compression and 1.08 for decompression, meaning that the total throughputs of the compressor and decompressor are increased by the same corresponding factors. However, the improvement is shared by the two data streams, so a single data stream achieves approximately 62% of the total throughput without time-sharing multi-threading for compression and 54% for decompression. Regarding the resource usage given in Table 16.2, the optimization reduces the combinational logic by 23–65%, the registers in the compressor module are increased by approximately 32%, and the number of registers is reduced to a third of that for the implementation without the optimization.

16.5 Related Works and Literatures

The most important lossless-compression algorithm is LZW, which is simple and effective and can be found in lossless-compression software such as gz, bzip2, rar, and lz. However, when attempting to implement a compressor on hardware, the problems discussed in this chapter inevitably arise. To implement compact hardware for LZW, we must prepare memory of the order of kilobytes. For example, Fowers et al. [3] and Kim et al. [5] solved the problem regarding the longest matching by parallelizing the operations. However, it is impossible to increase the size of the sliding dictionary because the number of start indices increases with the length of the symbols. Another important research topic is how to manage the symbol LUT in a limited memory space.

The field of machine learning contains well-known algorithms such as lossy counting [9] and the space saving [13]. However, these algorithms use operations based on pointers and are implemented in software. For a data stream with k different symbols, an attractive algorithm for frequency counting has been proposed in which

the top- θk frequent items are counted exactly within $O(1/\theta)$ space [4] for any constant $0 < \theta < 1$. However, this also provides a software solution. Various hardware implementations of lossless data-compression techniques have been investigated in this decade, and a well-known approach is *arithmetic coding* (here in short, AC) [6], which is used widely to compress multimedia data. Arithmetic coding includes heavy computation with floating-point numbers to achieve high compression ratios. To avoid floating-point calculations, arithmetic coding based on binary numbers has been proposed [1, 7, 15]. However, it is not possible to avoid the potential fractal computation, which is why hardware implementations such as those by Pande et al. [16] and Mitchell et al. [14] have been proposed to accelerate the computing speed.

References

1. T.D. Chuang, Y.J. Chen, Y.H. Chen, S.Y. Chien, L.G. Chen, Architecture design of fine grain quality scalable encoder with CABAC for H.264/AVC scalable extension. *J. Signal Proces. Syst.* **60**(3), 363–375 (2010)
2. Compressed Indexes and their Testbeds: <http://pizzachili.dcc.uchile.cl/> (2021)
3. J. Fowers, J.Y. Kim, D. Burger, S. Hauck, *A Scalable High-Bandwidth Architecture for Lossless Compression on FPGAs* (2015)
4. R.M. Karp, S. Shenker, C.H. Papadimitriou, A simple algorithm for finding frequent elements in streams and bags. *ACM Trans. Database Syst.* **28**(1), 51–55 (2003)
5. J.K. Kim, S. Hauck, D. Burger, A scalable multi-engine xpress9 compressor with asynchronous data transfer, in *IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (IEEE, 2014), pp. 161–164
6. G. Langdon, J. Rissanen, Compression of black-white images with arithmetic coding. *IEEE Trans. Commun.* **29**(6), 858–867 (1981)
7. C.C. Lo, S.T. Tsai, M.D. Shieh, Reconfigurable architecture for entropy decoding and inverse transform in H.264. *IEEE Trans. Consum. Electron.* **56**(3), 1670–1676 (2010)
8. K. Locke, Parameterizable content-addressable memory. Xilinx XAPP115 (2011)
9. G.S. Manku, R. Motwani, VLDB Endowment, Approximate frequency counts over data streams(2002), pp. 346–357
10. K. Marumo, S. Yamagiwa, *Time-Sharing Multithreading on Stream-Based Lossless Data Compression* (IEEE, 2017), pp. 305–310
11. K. Marumo, S. Yamagiwa, R. Morita, H. Sakamoto, Lazy management for frequency table on hardware-based stream lossless data compression. *Information* **7**(4), 63 (2016)
12. S. Maruyama, H. Sakamoto, M. Takeda, An online algorithm for lightweight grammar-based compression. *Algorithms* **5**(2), 214–235 (2012)
13. A. Metwally, D. Agrawal, A.E. Abbadi, An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Trans. Database Syst.* **31**(3), 1095–1133 (2006)
14. J.L. Mitchell, W.B. Pennebaker, Optimal hardware and software arithmetic coding procedures for the q-coder. *IBM J. Res. Dev.* **32**(6), 727–736 (1988)
15. R.R. Osorio, J.D. Bruguera, Arithmetic coding architecture for H.264/AVC CABAC compression system, in *Euromicro Symposium on Digital System Design, 2004. DSD 2004* (2004), pp. 62–69
16. A. Pande, J. Zambreno, P. Mohapatra, Hardware architecture for simultaneous arithmetic coding and encryption (2011)
17. J.S. Vitter, Design and analysis of dynamic Huffman codes. *J. ACM* **34**(4), 825–845 (1987)
18. S. Yamagiwa, K. Aoki, K. Wada, Performance enhancement of inter-cluster communication with software-based data compression in link layer. *Proc. IASTED PDCS* **2005**, 325–332 (2005)

19. S. Yamagiwa, K. Marumo, H. Sakamoto, Stream-based lossless data compression hardware using adaptive frequency table management, in *Proceedings of the VERY LARGE DATA BASES/BPOE 2015, LNCS 9495* (Springer, 2015)
20. S. Yamagiwa, H. Sakamoto, in *A Reconfigurable Stream Compression Hardware Based on Static Symbol-Lookup Table*. (IEEE, 2013), pp. 86–93
21. J. Ziv, A. Lempel, A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory* **23**(3), 337–343 (1977)
22. J. Ziv, A. Lempel, Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theory* **24**(5), 530–536 (1978)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.