Journal of ELECTRICAL ENGINEERING, VOL. 64, NO. 1, 2013, 44–49

A MULTI–ALPHABET ARITHMETIC CODING HARDWARE IMPLEMENTATION FOR SMALL FPGA DEVICES

Anton Biasizzo — Franc Novak — Peter Korošec *

Arithmetic coding is a lossless compression algorithm with variable-length source coding. It is more flexible and efficient than the well-known Huffman coding. In this paper we present a non-adaptive FPGA implementation of a multi-alphabet arithmetic coding with separated statistical model of the data source. The alphabet of the data source is a 256-symbol ASCII character set and does not include the special end-of-file symbol. No context switching is used in the proposed design which gives maximal throughput without pipelining. We have synthesized the design for Xilinx FPGA devices and used their built-in hardware resources.

K e y w o r d s: arithmetic coding, compression algorithm

1 INTRODUCTION

Arithmetic coding [1–3] is a lossless compression algorithm that can achieve entropy limit for a long data sequence. It uses a variable-length source coding and is superior to the well-known Huffman coding which uses fixed-length coding. It is also suitable for on-line implementation and the coding process can be separated from the source model. This allows concurrent adaptation of the source model which in turn leads to even better compression ratio.

Traditional coding techniques like Huffman coding replace the input symbol with its corresponding code. Such techniques can achieve entropy limit only when the probability of each input symbol is inversely proportional to the length of its code. Arithmetic coding uses the interval arithmetic while processing input symbols instead of using codes for each symbol. The coding starts with the coding interval $[0,1)$ and while each symbol from the input data stream is processed the coding interval is shrunk proportionally to the probability of the current symbol. The resulting interval uniquely represents the already processed input sequence. After all input symbols are processed the most suitable number from the coding interval is selected and it represents the complete input data stream.

The effectiveness of the arithmetic coding is dependent on the precision of arithmetic operations and the accuracy of the source model. The main disadvantage of the arithmetic coding is the use of expensive multiplications and in the case of adaptive source model even more expensive divisions. In the case of binary alphabet the arithmetic coding can be simplified and there are many hardware implementations mainly used in different multimedia codecs. There are also some multiplication-free implementations of the binary arithmetic coders [4–6] however they have lower compression ratio.

Multi-alphabet arithmetic coder can achieve higher data throughput and has a better compression ratio than binary arithmetic coder. While numerous software implementations [2, 3] of the multi-alphabet arithmetic coding algorithm have been reported, hardware implementations are fairly rare [9–11]. The main reason is the requirement of several (more than one) multipliers. Indeed, there are some multiplication-free implementations which typically use lookup tables for the multiplication but their compression efficiency is lower compared to the original algorithm.

The majority of hardware implementations of the multi-alphabet arithmetic coding algorithm [10, 11] use the property to convert multi-alphabet coding to binary coding using context switching. While this approach reduces the number of required multipliers the compression throughput reduces unless pipelining is used.

In this paper we present the hardware implementation of a multi-alphabet arithmetic coding algorithm using non-adaptive multi-alphabet source model. A 256-symbol ASCII character set is used as a data source. In contrast to [11] no context switching is used. In this way, a symbol is encoded at once, which gives maximal throughput without pipelining. We have implemented both arithmetic coder and decoder and evaluated their performance. Both designs were implemented on Xilinx Spartan3E xc3s500e device.

2 ARITHMETIC CODING

In arithmetic coding, a message is represented by an interval of real numbers in the range $[0,1)$. With each incoming symbol, the representing interval also referred to as current coding interval shrinks. The incoming symbol from the message reduces the size of the interval proportionally to the symbol probability. The symbols with

* Jožef Stefan Institute Jamova cesta 39, 1000 Ljubljana, Slovenia anton.biasizzo@ijs.si, franc.novak@ijs.si, peter.korosec@ijs.si

DOI: 10.2478/jee-2013-0006, ISSN 1335-3632 © 2013 FEI STU

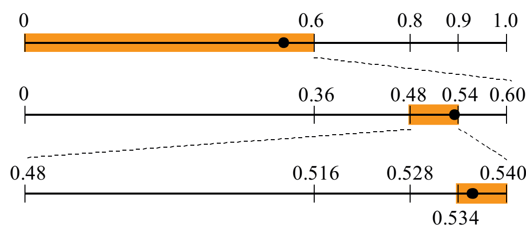

Fig. 1. Arithmetic coding process of the input message "ACD"

Table 1. Source model of the 4 symbol alphabet

Symbol s_i	Probability p_i	Cum. Prob. c_i	Subinterval
A	0.6	0	$[0, 0.6)$
B	0.2	0.6	$[0.6, 0.8)$
C	0.1	0.8	$[0.8, 0.9)$
D	0.1	0.9	$[0.9, 1)$

lower probability result in shorter intervals, which in turn require more bits in the encoded stream for their representation. Once all the input symbols are processed, any value from a determined subinterval can be chosen to represent the input message. The value is chosen such that it is described by a minimum number of bits.

Let the alphabet consist of symbols $s_0, s1, \ldots, s_{m-1}$, where m is the number of symbols in the alphabet. Their probability of occurrence is manifested in the source model: a semi-open interval $[0, 1)$ is divided into subintervals, each corresponding to a symbol in the input alphabet. The length of the subinterval is equal to $p_i = p(s_i)$, where $p(s_i)$ is the probability of occurrence of the symbol s_i. The left endpoint (start) of the interval c_i is equal to the sum of the probabilities of all symbols that precede the symbol s_i and is referred to as cumulative probability. It is given by

$$c_i = \sum_{j=0}^{i-1} p(s_j)\,. \tag{1}$$

Let us clarify this by an example. Let the input alphabet consist of the following symbols: A, B, C, D with the probabilities of their occurrence given in the second column in Tab. 1. The computed cumulative probabilities and the corresponding subintervals are given in the next two columns.

There are various methods to estimate the probabilities of the occurrence of input symbols and consequently to build the source model. The most straightforward and least precise is to estimate the probabilities from the predetermined symbol frequencies of the input language using given alphabet. A more accurate approach is to first determine the frequencies of each symbol in the message and calculate their respective probabilities. The disadvantage of this approach is that the massege must be stored and scanned in order to determine the source model.

Even better results can be achieved if higher order source models are used. These models apply conditional probabilities of the symbol based on the last few received symbols [11]. But they require substantial memory space and are impractical for hardware implementation.

Due to the nature of the arithmetic coding algorithm it is possible to change the source model after each symbol is processed. The adaptive source model [2] uses this property. In this approach, we start with a broad estimation and refine it by updating the probability of a symbol occurrence with each processed symbol. In this way, the source model adapts to the incoming message on the fly. The disadvantage of the adaptive source model is its computational complexity.

Our implementation uses static source model which can be determined either by basic estimation of the symbol probabilities or by counting the symbol frequencies in the message.

The process of arithmetic coding is briefly described below.

The received symbols (*ie*, the symbols received so far) determine the current coding interval. Let us denote the left endpoint of the current coding interval by the base or starting point b_i and the length of the coding interval by l_i. When a new input symbol s_k is processed, the coding interval is updated by the following recurrences

$$\begin{aligned} b_{k+1} &= b_k + c_k l_k \\ l_{k+1} &= p_k l_k\,. \end{aligned} \tag{2}$$

Initially the coding interval is typically set to the interval $[0, 1)$ hence b_0 is 0 and l_0 is 1.

Note that the principle of arithmetic coding can be used with any number interval as an initial interval if the same interval is used in the decoding process.

The coding interval could also be represented by its endpoints. Some, mainly software, implementations prefer this representation, however the recurrences must be modified and must include more arithmetic operations. Since the number of operations is critical for hardware implementation we have chosen the former interval representation.

The arithmetic coding process of a message "ACD" for the alphabet from the previous example is shown in Fig. 1.

After the complete message is coded the resulting subinterval represents the message. Any number from this interval can also be chosen to represent the message. From this interval we select the number with the shortest binary representation. This number is the code of the message.

In our example the final interval is $[0.534\,, 54)$ and the selected number, represented by a dot in Fig. 1, is 0.5390625. This number, written in binary number system, is 0.1000101_2 and is represented by 7 bits (1000101).

These bits are the code of the message “ACD” and 7 is the length of the code.

In practice, the coding algorithm is implemented in finite precision arithmetics. In this regard, the major concern is to ensure that the coding intervals do not overlap. Overlapping of the coding interval is prohibitive since different messages are represented by the same numbers in the cross section of the overlapped interval. The arithmetic coding process described by the previous equations assumes infinite arithmetic precision which assures distinct intervals.

The implementation of the coding algorithm must consider the given precision. A. Said [2] showed that distinct intervals are assured also with finite precision arithmetics. In fact, the rounding errors due to the finite arithmetics can be interpreted as the inaccuracy of the input model. The resulting encoded message is still decodable but its compression ratio is slightly below the entropy based limit.

There is another problem due to finite precision arithmetic. When the coding interval becomes small, it cannot be further divided into sub-intervals. At this point the leading portions of the numeric format of the endpoints become equal and cannot change during further coding process. This leading portion represents the leading portion of the code.

Problems with small coding intervals can be circumvented by the fact that any interval can be used for coding thus when coded interval becomes small we can scale it up. It was shown [2] that the coding interval can be scaled up and/or moved by any number without affecting the coding process as long as the same scale and move operations are performed in the decoding process. Using this property it is easy to maintain the appropriate accuracy by applying integer arithmetics. This process is called re-scaling.

As mentioned previously, the selection of the initial interval and the use of re-scaling process does not affect the coding process. However they have impact on the accuracy of the operations and consequently on the compression ratio. To maintain high accuracy the length of the interval should be kept as high as possible.

The decoding process is almost identical to the coding process. It processes current coding interval but in addition it maintains also the coded message. It has to start with the same initial interval and perform identical operations as the coding process (update and rescale operations). In addition it has to search the source model to find out in which subinterval the coded message lies. The search can be accomplished by traversing the symbols in one direction or by bisection, which is quicker but more difficult to implement. Because of the search the decoding process is inherently slower than the coding process.

3 FPGA IMPLEMENTATION

Hardware implementations of arithmetic coding are mostly limited to binary arithmetic coding. The main reasons are relatively complex arithmetic operations (*eg* multiplication). The input alphabet of binary arithmetic coders consists of only two symbols and the algorithm can be implemented without multiplication [4, 5]. Hardware binary arithmetic coders are commonly used in different multimedia codecs to accelerate the processing of video or audio streams.

The multi-alphabet arithmetic coding can be implemented using binary arithmetic coding. In this case, the multi-alphabet is transformed to a binary alphabet applying arbitrary fixed-length binary coding. If the new binary source model is fixed then the overall compression ratio is lower than the original one. The reason for this lies in the fact that the computed probability of binary alphabet does not reflect the actual probability distribution of individual bits of a symbol. To overcome this drawback the binary source model is switched to the binary probability distribution that corresponds to the currently processed bit of the input symbol code. This process is referred to as context switching. The majority of multi-alphabet hardware implementations use this approach [10, 11]. This, however, is relatively cumbersome and also slower then the original multi-alphabet implementation since only one bit can be processed within a single clock cycle. Our implementation does not use context switching but rather apply multiplications on the original probability distributions of the multi-alphabet.

While multiplication still represents a notable cost in ASIC implementations the majority of FPGA devices already have built-in hardware multipliers. The use of these devices does not increase the cost of the overall design as long as there are enough multipliers at disposal in the given FPGA device.

3.1 Arithmetic coder

The use of built-in multipliers enabled us to adopt efficient software solutions of the multi-alphabet arithmetic coding [2, 3] to our FPGA-based hardware implementation. In our implementation, the source model is separated from the coding process as depicted in Fig. 2. In order to spare hardware resources, static source model is employed. As noted in the introduction, the adaptive source model requires division operation, which is more expensive than multiplication and requires several clock cycles per division.

In order to minimize the number of mathematical operations, a coding interval is represented by its base and its length. They are stored in binary registers with a given precision whose values are interpreted as fractions. The coding interval is re-scaled if its length is smaller than 0.5, which is a half of the length of the initial coding interval. This yields better accuracy because the length of the current interval is kept as large as possible, which in turn decreases the relative rounding errors. This way the base of the rescaled current coding interval lies on the interval $[0, 1)$ while the length lies on the interval $(0.5\,,\,1]$. Thus the endpoint lies on the interval $(0.5\,,\,2)$ which is

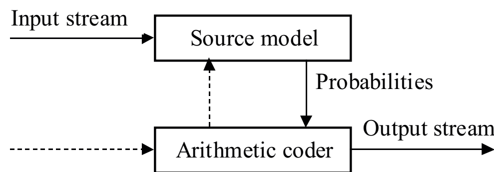

Fig. 2. General structure of the arithmetic coding

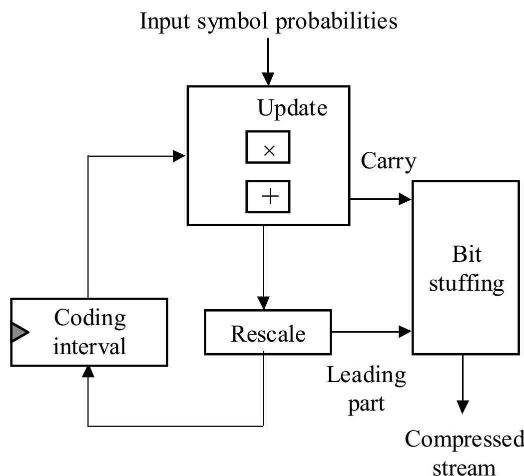

Fig. 3. The structure of the arithmetic coding unit

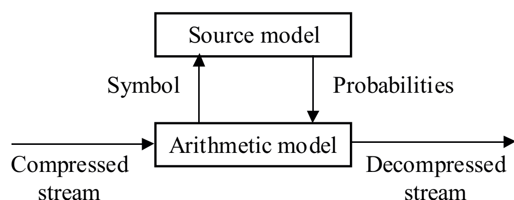

Fig. 4. General structure of the arithmetic decoding

larger then the initial coding interval $[0, 1)$ and the end-point may fall above 1. This introduces the problem of carry propagation. During re-scaling, bits are shifted out from the base of the current interval and represent the leading part of the base, which represents the leading portion of the coded message. If the current coding interval is a sub-interval of $[0, 1)$ then this leading part is settled. Otherwise it may occur that in the subsequent interval update operation the current coding interval, *ie* the base of the current coding interval, falls above 1. In such cases the interval is moved left by 1 and the leading part of the base is incremented, which is called carry propagation. The leading part is held in a buffer and it can be output when it becomes settled. The addition of carry is handled by buffering only the last nonzero bits of the leading part of the base with their preceding zero bit. In the case of carry these bits are negated and output because they become settled. More efficient buffering can be implemented by counting the last nonzero bits. In both cases there can be a problem when allocated resources are exhausted. This is called carry-over problem and is resolved by outputting the buffer and adding a 0 in the output stream. This technique is known as bit-stuffing [2].

Our implementation uses the counter of the nonzero bits in the output stream since it consumes less hardware resources and if the counter is big enough the probability of the carry-over situations becomes negligible. Nevertheless we included the bit-stuffing logic to eliminate potential problems with coding of very long messages.

The hardware implementation is depicted in Fig. 3 and consists of the following modules:

- Interval update module, which multiplies the length of the coding interval with both the probability and the cumulative probability of the input symbol. It updates the base and the length of the coding interval with these products as defined by Equation 2. In the case when the new base lies above 1 the coding interval is moved left by 1 and the carry bit is set. The carry bit is fed to the bit-stuffing module.
- Re-scale module, which shifts both the base and the length of the interval left (multiplication by the 2) until the length of the coding interval is bigger than one half of the maximal length of the coding interval. This is detected by monitoring the most significant bit of the length of the current interval. The bits that are shifted out are collected by the bit-stuffing module.
- Bit-stuffing module, which collects the bits from the re-scale module into a buffer of the leading part of encoded message, counts the last non-zero bits and outputs the settled part of the buffer. If the carry bit from the update module is active it increments current buffer and flushes the buffer. In the case of carry-over situation, it flushes the buffer and initialize buffer to a single bit with value 0.

3.2 Arithmetic decoder

The general structure of the arithmetic decoding is shown in Figure 4. As in the case of the arithmetic coder, the source model is separated from the decoder. If the adaptive source model is used, the updating must resemble the update of the source model in the arithmetic coder. In our implementation the source model is static.

The structure of the arithmetic decoder resembles the structure of the arithmetic coder: it has the same update interval module and the rescale module since it has to perform exactly the same operations. The main difference is that from the encoded stream it determines the current coded value used to select the corresponding symbol. Since there is no direct mapping between the current coded value and the correct symbol, the source model must be searched to identify the symbol. Additionally, at the input of arithmetic decoder there is a bit-stuff detection module.

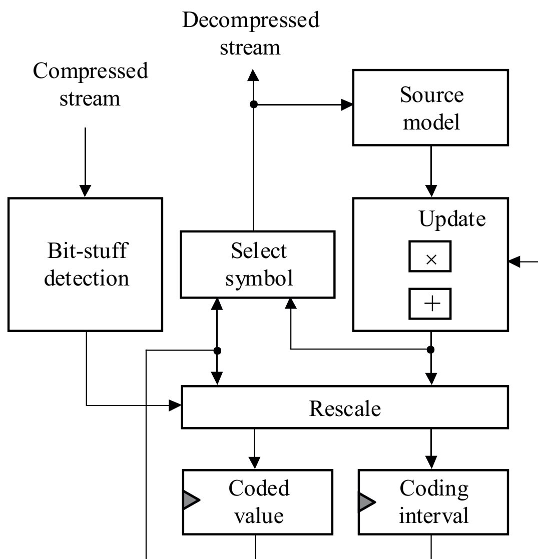

Fig. 5. The structure of the arithmetic decoder with the external source model

Table 2. The usage of FPGA resources for arithmetic encoder

Resource	Used	Utilization
Slice Flip Flops	114	1%
4 Input LUTs	630	6%
Slices	339	7%
BRAMs	1	5%
Multipliers	2	10%

Table 3. The usage of FPGA resources for arithmetic decoder

Resource	Used	Utilization
Slice Flip Flops	120	1%
4 Input LUTs	392	4%
Slices	208	4%
BRAMs	1	5%
Multipliers	1	5%

The hardware implementation is depicted in Fig. 5 and consists of the following modules:

- Interval update module, which is the same as the interval update module of the arithmetic coder.
- Re-scale module, which resembles the re-scale module of the arithmetic coder. In addition, it has to re-scale (shift) the current coded value needed for symbol selection.
- Bit-stuff detection module, which detects the stuffed bits and removes them from the encoded stream.
- Symbol selection module, which performs the search of the correct symbol over the source module using the bisection method. For a 256 symbol alphabet it needs 8 cycles to determine the correct symbol. The search is performed when the coding interval is as large as possible, *ie* after it is re-scaled.

3.3 Source model

The static source model is implemented as a lookup table. Arithmetic coder and decoder require two values associated with the current symbol: either symbol probability and its cumulative probability or two consecutive cumulative probabilities. This can be achieved with two separate single-port RAMs or with one dual-port RAM.

In our implementation we used dual port BRAM resources for storing the cumulative symbol probabilities. The probability values are loaded into the BRAM before the encoding or decoding process is started.

3.4 Implementation details

Hardware arithmetic coder was implemented for a 256-symbol source model using 12 bit precision and 12 bit precision fixed point arithmetics. The bit-stuffing also used 12 bit counter for counting non-zero output bits. Both devices were synthesized separately on Xilinx Spartan3E xc3s500e device [7] using Xilinx ISE 13.1 development environment. Table 2 summarizes the usage of the FPGA resources for the arithmetic encoder.

Table 3 gives the usage of the FPGA resources for the arithmetic decoder.

The implementation of the arithmetic encoder as well as the implementation of the arithmetic decoder have minimum clock period below 10 ns. Since the output of the arithmetic coder is 1 bit wide the ideal data throughput of the arithmetic coder can be up to 100 M bps. The actual data throughput is slightly lower due to the update process. A data throughput of 88 M bps was achieved during the compression of some generic data stream.

The upper bound of the data throughput of the arithmetic decoder is also 100 M bps since during each clock cycle a single bit can be processed at the input. The actual throughput of the arithmetic decoder is lower than the data throughput of the arithmetic coder because at each update interval operation a symbol search must be performed. The symbol search requires 8 iterations. A throughput of 47 M bps was achieved during the decompression of previously compressed data stream.

4 CONCLUSION

Both designs can be implemented in a very small FPGA device and are suitable also for embedded applications. The througput of both designs can be improved by using the wider output ports and using barrel shifter for the rescale operation. Their low FPGA device utilization also enables either parallel implementation of coding and decoding devices (*eg* for compression of LIDAR data [8])

or for pipelined implementation of the arithmetic coding to improve the compression throughput.

Acknowledgments

This paper has been developed within activities of the project "Processing of massive geometric LIDAR data", L2-3650, supported by the Slovenian Research Agency.

References

[1] LANGDON, G. : An introduction to Arithmetic Coding, IBM J. Res. Develop. **28** (Mar 1984), 135–149.

[2] SAID, A. : Introduction to Arithmetic Coding Theory and Practice, Hewlett-Packard Laboratories Report, HPL-2004-76, April 2004.

[3] WITTEN, I. H.—NEAL, R.—CLEARY, J. G. : Arithmetic Coding for Data Compression, Communications of the ACM **30** (June 1987), 520–540.

[4] ANDRA, K.—ACHARYA, T.—CHAKRABARTI, C. : A Multi-Bit Binary Arithmetic Coding Technique, In: Proceedings Int. Conf. Image Process., vol. 1, 2000, pp. 928-931.

[5] HAI, M.—ZHANG, J. J.—NI, X. F. : An Improved Arithmetic Coding Algorithm, Journal of Shanghai University (English Edition) **8** No. 4 (2004), 455–457.

[6] RISSANEN, J. MOHIUDDIN,—K. M. : A Multiplication-Free Multialphabet Arithmetic Code, IEEE Transactions On Communications **37** No. 2 (Feb 1989), 93–98.

[7] XILINXCORP : Xilinx DS312 Spartan-3E FPGA Family Data Sheet, electronic file available at http://www.xilinx.com/ support/documentation/data_sheets/ds312.pdf, July 2011.

[8] MONGUS, D.—ŽALIK, B. : Efficient Method for Lossless LIDAR Data Compression, International Journal of Remote Sensing **32** No. 9 (2011), 2507–2518.

[9] PRINTZ, H.—STUBLEY, P. : Multialphabet Arithmetic Coding at 16 MBytes/sec, In: Proceedings Data Compression Conference, 1993, pp. 128–137.

[10] MAHAPATRA, S.—SINGH, K. : A Parallel Scheme for Implementing Multialphabet Arithmetic Coding in High-Speed Programmable Hardware, In: Proceedings Int. Conf. on Information Technology: Coding and Computing, ITTC'05, vol. 1, 2005, pp. 79–84.

[11] MAHAPATRA, S.—SINGH, K. : An FPGA-Based Implementation of Multi-Alphabet Arithmetic Coding, IEEE Trans. On Circuits and Systems I **54** No. 8 (2007), 1678–1686.

Received 25 April 2012

Anton Biasizzo has been a researcher at the Joef Stefan Institute since 1991. He received a PhD degree from the University in Ljubljana in 1998. His research interests include efficient algorithms for sequential diagnosis, constraint logic programming, model-based diagnosis and automatic test-pattern generation.

Franc Novak gained his PhD degree in electrical engineering from the University in Ljubljana in 1988. Since 1975 he has been with the Jožef Stefan Institute, where he is currently head of the Computer Systems Department. He is also full professor at the Faculty of Electrical Engineering and Computer Science, University of Maribor. His research interests are in the areas of electronic testing and diagnosis, and fault tolerant computing. His most recent assignment has been on design of built-in self-test of reconfigurable systems.

Peter Korošec received his PhD from the Jožef Stefan International Postgraduate School, Ljubljana, Slovenia, in 2006. Since winter 2002 he has been a researcher at the Joef Stefan Institute, Ljubljana, Slovenia. He is presently a researcher at the Computer Systems department and an associate professor at the University of Primorska, Faculty of Mathematics, Natural Sciences and Information Technologies Koper, Koper, Slovenia. His research interests include combinatorial and numerical optimization with modern metaheuristics in parallel and distributed computing.

EXPORT - IMPORT
of periodicals and of non-periodically
printed matters, books and CD-ROMs

Krupinská 4 PO BOX 152, 852 99 Bratislava 5, Slovakia
tel: ++421 2 638 39 472-3, fax: ++421 2 63 839 485
info@slovart-gtg.sk; http://www.slovart-gtg.sk