Cezary WERNIK, Grzegorz ULACHA
WEST POMERANIAN UNIVERSITY OF TECHNOLOGY SZCZECIN
FACULTY OF COMPUTER SCIENCE AND INFORMATION TECHNOLOGY
52 Żołnierska St., 71-210 Szczecin

An implementation of Rice coder on AVR platform

Abstract

In this paper the features of the Rice code and its performance by looking at the requirements for hardware resources are presented. The estimated bandwidth of the encoder was examined. An example implementation of the Rice coder on the AVR platform was also presented, using the Arduino UNO boards for this.

Keywords: Rice code, Arduino application for coding, AVR, programmable devices.

1. Introduction

Universal codes are used to represent sources with an infinite number of symbols, most often treated as a set of positive natural numbers n [7]. For some applications, this set also includes zero. Until designing codes with an infinite number of source symbols, two general assumptions are made. Firstly, the coding method should be able to be described in the form of an algorithm without the use of tables containing code words (in the case of codes with a finite number of symbols, the source code words can be described using a table or e.g. binary trees). Secondly, $p(m) \geq p(n)$ for $m < n$. Depending on the source distribution, different codes are used. The simplest of codes is the Elias code family [7]. They are a code α (unary code), a code β (binary code) and a few defined in successive letters of the Greek alphabet. Elias and Levenstein described in the works [1], [3] variations of gamma codes (γ, γ'). The γ' code is sometimes referred to as the Exponential Golomb code [5] and is used for e.g. in the tenth part of the MPEG4 standard referred to as H.264.

In 1966 Solomon W. Golomb presented in [2] the basic assumptions and examples of the family of codes which were described as Golomb codes. This universal code is used to represent non-negative integers n, whose probability of occurrence is consistent with the geometric distribution $G(n) = (1-p) \cdot p^n$. For such a distribution after selecting the appropriate parameters (based on value the parameter p) the Golomb code becomes a code convergent with the Huffman code, and this one meets the definition of the optimal code among the codes described using binary trees [6].

Assuming the constant value of the parameter p algorithm for determining the Golomb code words is quite simple although it requires the use of multiplication and division operations. After introducing further simplifications in the algorithm is possible to get rid of the multiplication and division operations. Such features have Rice's code being a special case of Golomb codes [4], [7].

2. Features of Rice code

With the geometric distribution we deal e.g. in the coding of prediction errors in systems of lossless compression of images and audio signals [8], [9]. The second popular example is the binary source $S = \{0, 1\}$ in which the stream of zeros and ones can be interpreted as a set of words in unary code (sequences of zeros terminated by one assuming that there are more zeros than ones, then $p = p(0) \geq p(1)$). For example, bit sequence 00100001011 can be converted to four natural numbers {2, 4, 1, 0} describing individual unary words. Each of these numbers can be coded in a highly efficient way using the Rice code. This second type of data will be further considered in this work.

After encoding such a binary input stream with the Rice code we get the output bit sequence in which its average number of bits L_R per one bit of input data is given by the formula:

$$L_R = (1-p) \cdot \left(k + \frac{1}{1-p^{2^k}} \right), \tag{1}$$

where the parameter k being a non-negative integer is called a group number which is determined in the form of the probability function p:

$$k = \left\lceil \log_2 \frac{\log_2 \frac{\sqrt{5}-1}{2}}{\log_2 p} \right\rceil. \tag{2}$$

The p-value ranges for the consecutive group numbers $k \leq 8$ are shown in Table 1. In Figure 1 the dependence of the mean L_R of the Rice code in function p are presented.

Tab. 1. The p-value ranges for consecutive group numbers $k \leq 8$

k	p
0	0,50000 — 0,61803
1	0,61803 — 0,78615
2	0,78615 — 0,88665
3	0,88665 — 0,94162
4	0,94162 — 0,97037
5	0,97037 — 0,98507
6	0,98507 — 0,99251
7	0,99251 — 0,99625
8	0,99625 — 0,99812

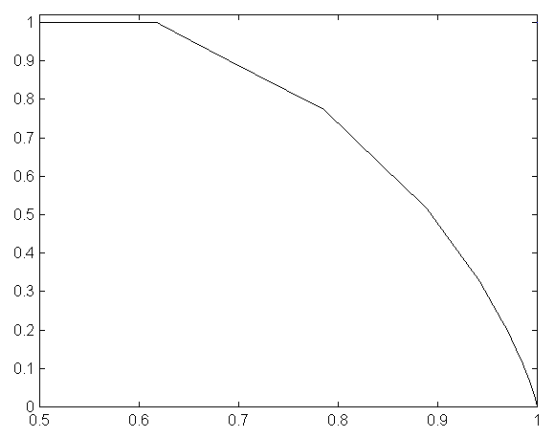

Fig. 1. Dependence of the mean L_R of the Rice code to the p function

We can calculate the percentage efficiency of the Rice code as the ratio of binary entropy to the average L_R:

$$E_R = \frac{H_{01}}{L_R} \cdot 100\% = \frac{-p \cdot \log_2 p - (1-p) \cdot \log_2 (1-p)}{L_R} \cdot 100\%. \quad (3)$$

The use of both Rice code and Golomb code to encode binary sources (understood as a set of independent symbols) guarantees high compression efficiency exceeding 95.94%. Figure 2. shows the dependence of the percentage E_R efficiency of the Rice code in the function of the probability p. As its show, local minimal values mean the transition (marked with vertical dashed lines) between consecutive group numbers k. To keep the drawing legible for $k > 3$ ($p > 0{,}94162$) dashed lines have not been placed.

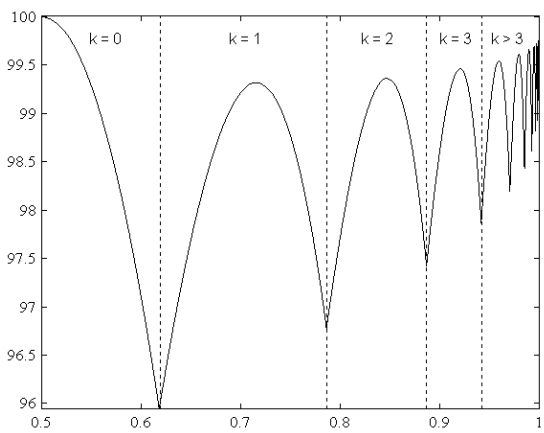

Fig. 2. The dependence of the percentage efficiency of Rice's code in function p

3. Implementation of the Rice code

Rice's word code consists of a prefix (in a unary code form) being the number of the group u and item number of the element in the group v represented as k-bits binary number (for $k > 0$). For the coded symbol n the group number is calculated from dependence:

$$u = \left\lfloor \frac{n}{2^k} \right\rfloor, \quad (4)$$

and the item number in the group (for $k > 0$):

$$v = n \bmod 2^k = n - u \cdot 2^k. \quad (5)$$

In a practical implementation the number u is obtained by a bit shift, and the number v is taken as k the less significant bits of the coded value n. The length of the Rice code word for the value of n expressed in bits is $l_n = u + k + 1$.

A ready formula can be useful for parallelising procedures of coding. In a situation where we do not have the parameter p (e.g. in prediction errors coding when the characteristics of the data distribution change over time) it is good to divide the coded stream into data packets each of which can be encoded with an individual parameter k. Its optimal selection requires finding the shortest encoded bit string result, with no need to perform a full encoding process for subsequent values of k from $k_{\min}$ to $k_{\max}$. It is enough to use the formula $l_n = u + k + 1$, and the procedure can be parallelised for individual k values.

For example, the data is binary source S_{01} where $p = p(0) = 0.8$. If we build the Rice code for this source, then on the basis of formula (2) we get $k = 2$. This gives an average $L_R = 0.73875$ and an $E_R = 97.722\%$ efficiency. Table 2. shows the first 10 words of the Rice code for $p = 0.8$ ($k = 2$), along with other k values: $p = 0.7$ ($k = 1$), $p = 0.9$ ($k = 3$), $p = 0.95$ ($k = 4$).

Tab. 2. Sample Rice code words at $k = \{1, 2, 3, 4\}$

n	Rice code words			
	k=1	k=2	k=3	k=4
0	1:0	1:00	1:000	1:0000
1	1:1	1:01	1:001	1:0001
2	01:0	1:10	1:010	1:0010
3	01:1	1:11	1:011	1:0011
4	001:0	01:00	1:100	1:0100
5	001:1	01:01	1:101	1:0101
6	0001:0	01:10	1:110	1:0110
7	0001:1	01:11	1:111	1:0111
8	00001:0	001:00	01:000	1:1000
9	00001:1	001:01	01:001	1:1001

4. An implementation of encoder on AVR platform

To show the simplicity and efficiency of the encoder and low hardware requirements as implementation environment for the encoder one of the popular AVR platforms has been chosen. To implementation was used Arduino UNO as encoder the bit stream in an asynchronous manner. Arduino is an open-source-hardware platform based on easy-to-use hardware and software [11]. On the Arduino board the microcontrollers from the AVR family are mounted. The microcontroller has a loaded bootloader, as the basic firmware, to which the user can add so-called "sketches" comprising the setup( ) and loop( ) functions, which in turn are used to implement basic microcontroller settings and implement the program appropriate for the user. The sketch is written in the high-level language C wrapped by the Wiring library. The Arduino UNO boards are equipped with a USB port, that is used as a serial port and a port for programming [10].

Two Arduino UNO were used for the implementation, one as the transmitter the second transmitter as a receiver. On this Arduino model, the ATmega328p microcontroller is mounted. Which achieves throughput approaching 1 MIPS per MHz [12]. Where the fastest assembler instruction is done in one clock cycles. Arduino UNO has an ICSP port and several digital ports, one of which can be used as an analog port (A/D).

The SD card was used as the data source. It has been connected to the transmitting Arduino UNO module and read by the SPI via ICSP port. Between the Arduino boards, asynchronous communication has been made using three digital pins.

One of the pins (pin 3) is used for data transfer (0/1 states), the other of pins (pin 4) is used to inform the receiving system that the data has been given, the third pin (pin 5) is used to receive feedback from the receiver, that he received the information (asynchronous transmission with confirmation).

In addition, pins 2, 7-10 were used for diagnostic purposes and coder status check. Pin 2 to inform if the encoder is in a data coding state, pin 7 to connect a green diode indicating that the encoder has terminated work, pin 8 to connect the red diode informing that the encoder is busy, pin 9 to inform if the encoder is transmitting data to the receiver, pin 10 to inform if the transmitter is reading data from the SD card. This is presented in Figure 3.

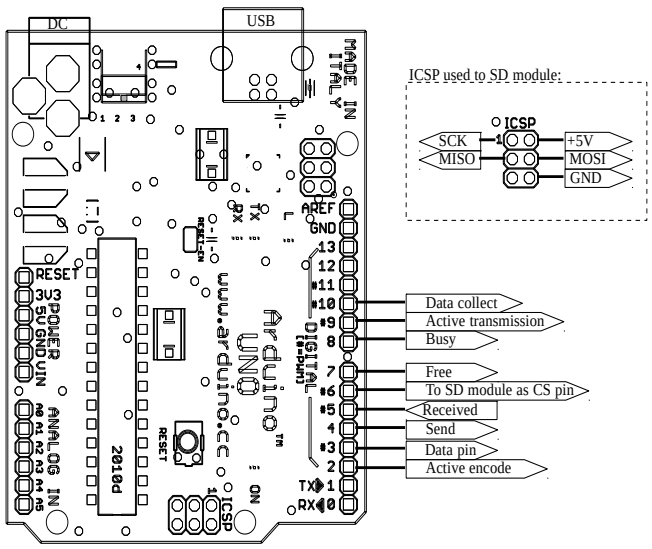

Fig. 3. Connection Arduino UNO boards and SD card module ([10] and own interpretation)

The cycle of reading data, data collection, encoding and transmission is as follows. The transmitter reads data from the SD card and collects the number of read zeros until it read a value of one of the input stream. After reading the value one, the transmitter codes the number of reads zeros *n* as numbers *u* and *v*, then the number *u* sends to the receiver as a unary sequence represented by zeros terminated by one. Then the number *v* sends to the receiver as a binary number written on *k*-bits. The full one cycle is presented in Figure 4.

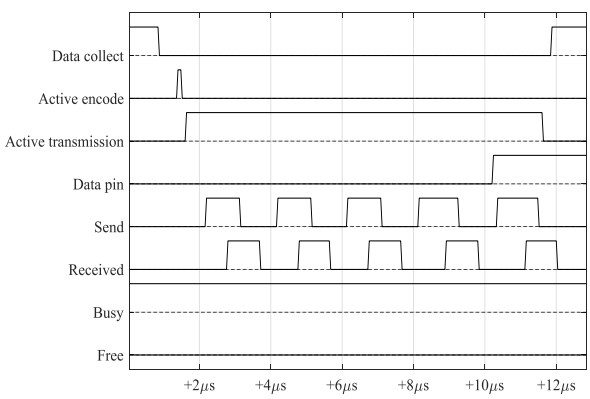

Fig. 4. Waveform of a single cycle of data collection, coding and transmission, data collected using Saleae Logic 1.2.18

The measurements carried out using of the logic state analyzer Saleae Logic 1.2.18 indicate that the data coding time depends on the group number *k.* The larger the value of *k*, the longer the calculation of *u* and *v* takes longer. However, in relation to the duration of asynchronous transmission and the time of reading data from the SD card, the coding time is small.

Figure 5. presents a comparison of the average duration of selected stages of the encoder's work, measured for data with different probabilities *p*.

The average coding time of a single received zero series is 0.820 microseconds, where the asynchronous transmission time achieved as fast as possible is 8.545 microseconds. The transmission time of a single bit is 1.657 microseconds, while the time of handling a single reading of the smallest data unit from an SD card is up to 33.549 microseconds.

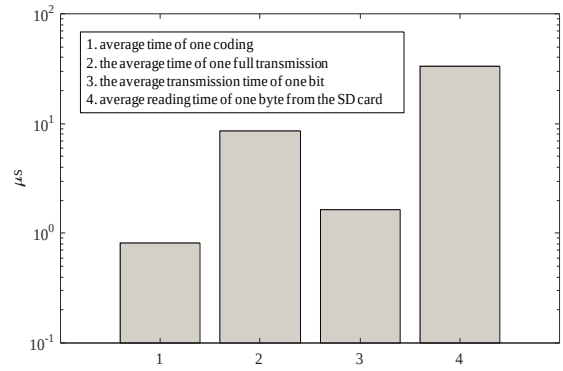

Fig. 5. Comparison of the average duration of selected stages of the coder operation

For correct coding, not including transmission, only one variable of 8 bit (unsigned char for *v*) and two 16 bit (unsigned int for *u* and *n*), but assuming the known probability *p* the implementation of calculation operations *u* and *v* can be closed in three variables 8 bits.

Due to the extensive library for handling the SD card, the Arduino sketch with the bootloader consumes 8488 B (26%) of program memory, where the maximum is 32256 B. All global variables including auxiliary variables consume 836 B (40%) of dynamic memory, leaving 1212 B for local variables, where the maximum is 2048 B.

5. Conclusions

The study shows that the Rice code keeps high efficiency and coding rate even with small hardware resources.

In our implementation, the encoder bandwidth is on the level up to 1.16-1.71 MB / s, which is significantly limited by the speed of the input stream and output stream, i.e. SD implementation and asynchronous output transmission proved to be the bottleneck of the implementation, not the compression process itself data.

6. References

[1] Elias P.: Universal codeword sets and representations of the integers, IEEE Transactions on Information Theory, March 1975, vol. 21, pp. 194-203.

[2] Golomb S. W.: Run-length encoding, IEEE Transactions on Information Theory, July 1966, vol. 12, pp. 399-401.

[3] Levenstein V. E.: On the redundancy and delay of separable codes for the natural numbers, Problems of Cybernetics, 1968, vol. 20, pp. 173-179.

[4] Rice R. F.: Some practical universal noiseless coding techniques, Jet Propulsion Labolatory, JPL Publication 79-22, Pasadena, CA, March 1979.

[5] Richardson I. E. G.: H.264 and MPEG4 video compression. Video coding for next-generation multimedia, West Sussex, England, John Wiley & Sons Ltd. 2004.

[6] Sayood K.: Introduction to Data Compression, ed. 2, Morgan Kaufmann Publ., San Francisco, 2002.

[7] Sayood K. (editor): Lossless compression handbook, California, Academic Press USA 2003.

[8] Ulacha G., Stasiński R.: Performance Optimized Predictor Blending Technique For Lossless Image Coding, in: Proceedings of The 36th International Conference on Acoustics, Speech and Signal Processing ICASSP'11, Prague, Czech Republic, 22-27 May 2011, pp. 1541-1544.

[9] Ulacha G., Stasiński R.: Entropy Coder for Audio Signals, International Journal Of Electronics And Telecommunications, 2015, Vol. 61, No. 2, pp. 219-224.

[10] https://www.arduino.cc/

[11] https://www.oshwa.org/definition/
[12] http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf

Received: 09.03.2018 *Paper reviewed* *Accepted: 04.05.2018*

Cezary WERNIK, MSc, eng.

Cezary Wernik received the MSc degree in computer science from the Szczecin University of Technology in 2017. His research interests concerns audio compression, music information retrieval (MIR) and embedded systems programming.

e-mail: cwernik@wi.zut.edu.pl

Grzegorz ULACHA, PhD, eng.

Grzegorz Ulacha (M'2000, PhD'2004) graduated from the Szczecin University of Technology, where he also defended his PhD thesis. He is working now as an assistant professor in Faculty of Computer Science & Information Technologies, West Pomeranian University of Technology, Szczecin. His scientific interests are mainly linked with lossless and lossy image and audio coding.

e-mail: gulacha@wi.zut.edu.pl