

Rice-Marlin Codes: Tiny and Efficient Variable-to-Fixed Codes

Manuel Martinez* and Joan Serra-Sagristà†

*Karlsruhe Institute of Technology
Karlsruhe, 76131, Germany
manuel.martinez@kit.edu

†Universitat Autònoma de Barcelona
Cerdanyola del Vallès, 08193, Spain
Joan.Serra@uab.cat

Abstract

Marlin [1, 2] is a Variable-to-Fixed (VF) codec optimized for high decoding speed through the use of small sized dictionaries that fit in the L1 cache of most CPUs. While the size of Marlin dictionaries is adequate for decoding, they are still too large to be encoded fast. We address this problem by proposing two techniques to reduce the alphabet size. The first technique is to encode rare symbols in their own segment, and the second is to combine Marlin dictionaries with Rice encoding, hence our name Rice-Marlin for our new codec. Using those techniques, we are able to reduce the size of Marlin dictionaries by a factor of 16, not only enabling faster encoding speed, but also achieving better compression efficiency.

1 Introduction

High Throughput (HT) codecs prime speed over compression efficiency. While HT codecs have been used to reduce storage needs (*i.e.*, LZ4 [3], and LZO [4]), their main application is to improve the throughput of communication interfaces [5].

Most generic lossless compression codecs that are not considered to be HT are based on the deflate [6] algorithm, which is a two-step algorithm that combines dictionary compression (*i.e.*, LZ77 [7]), and an entropy encoder (*i.e.*, Huffman [8], arithmetic [9] or range encoding [10]). However, entropy encoding is significantly slower than the dictionary compression step and, furthermore, in most target applications, entropy coding provides reduced compression gains as compared to those due to dictionary compression. Hence, in most HT algorithms, the entropy encoding step is either dropped [3, 5] or severely simplified [4, 11]. In particular, most research in HT focuses on finding novel ways to find matches in LZ77 dictionaries that offer compelling tradeoffs between coding speed and efficiency [12, 13].

Still, there are applications that need a fast entropy codec. In particular data obtained by sensors (*i.e.*, image, audio, etc.) generally can not be compressed well using generic dictionary approaches; hence, LZ77 based HT codecs do not perform well in this kind of data. One option to deal with this problem is to use fast versions of generic entropy codecs (*i.e.*, Huff0 [14] for Huffman and FSE [15] for Asymmetric Numeral Systems [16]). However, if we know that the source we are compressing

JSS acknowledges support from Spanish Ministry of Economy and Competitiveness (MINECO) and European Regional Development Fund (FEDER) under Grant TIN2015-71126-R, and from Catalan Government under Grant 2017SGR-463. This work is also partially supported by the German Federal Ministry of Education and Research (BMBF) within the KonsensOP project.

provides symbols following a particular probability distribution, we can use a codec that specifically matches this distribution. The typical example of this case is when encoding sensor residuals, which often follow Laplacian distributions. For this particular distribution, the Rice codec [17] is close to optimal, and faster than generic entropy codecs. Hence, Rice encoding is very popular in lossless image codecs [18, 19].

Rice codec is a Fixed-to-Variable (FV) codec, where each input symbol is encoded as a variable number of bits. Generally, FV codecs can be encoded exceedingly fast using a lookup table, but are slower to decode as the boundaries between compressed symbols are not known in advance. On the other hand, Variable-to-Fixed (VF) codecs (*e.g.*, Tunstall [20]) can be decompressed fast, but are slower to compress.

Marlin [1, 2] was the first VF codec that achieved a competitive compression ratio at HT decoding speeds thanks to the combination of using plurally parsable dictionaries [21, 22, 23, 24, 25] in a memoryless context, however, being a VF codec, Marlin is slow to encode. The main bottleneck during the encoding process is related to the size of the dictionary. For decoding, we only need to store the dictionary itself on the L1 cache, but for encoding, we need to encode either a tree or a hash table whose size is several times larger than the dictionary itself.

In VF codecs, the size of the dictionary must be larger than the alphabet size. Hence, when encoding 8-bit sources the size of the dictionary must be larger than 256 entries. As the next efficient value for the codeword size is 12-bit, Marlin dictionaries have 4096 entries. This size poses two problems: first, the encoding tree does not fit in the L1 cache, and also 4096 entries are not enough for efficient encoding of high entropy sources. We deal with those two problems in this contribution.

Here we apply two techniques that reduce the number of symbols to be encoded by Marlin, effectively decreasing the size of the Marlin alphabet as much as possible.

Our first technique is to remove non-compressible bits (*i.e.*, bits whose probability of being 0 and 1 is close to 50%, and are uncorrelated to other bits). Having non-compressible bits in an alphabet has an extremely negative effect on its efficiency, and it is thus worth just to store them in a binary form, similarly to what Rice codes do. Hence we named the combination Rice-Marlin codec. Each stored bit halves the alphabet size, duplicating the ratio between dictionary size and alphabet size.

Our second technique consists of removing from the alphabet those symbols whose probability of being emitted in a message is below a certain threshold, becoming then unrepresented symbols. Each occurrence of an unrepresented symbol is stored independently in a dedicated section.

The first technique is more effective when compressing high entropy sources, while the second technique is more effective when compressing low entropy sources. By combining both techniques we are able to use 8-bit codewords along all the entropy ranges, while improving the compression ratio over previous approaches.

Experimental results for both synthetic and real data reveal how Rice-Marlin codes achieve better compression ratio, the same decoding speed, and 2.4x faster encoding speed when compared to original Marlin codes.

2 Background: Marlin Dictionaries

In this section we review the concepts from Marlin dictionaries that are necessary to present our new contributions.

Marlin expects an alphabet A that contains a certain number of symbols and their probabilities, and builds a dictionary \mathcal{W} with 2^N words. Most VF codecs do uncompress a message by consuming N bits from the compressed stream and emit its corresponding word. Marlin uses instead an overlapping codeword methodology [2], where N bits are *peeked* from the stream, corresponding to one entry in the dictionary, but only K bits are consumed, as seen in Fig. 1. In our notation, K is the number of bits that are consumed from the source at each iteration, and O is the number of bits that overlap between codewords, with $N = O + K$.

0000: aaaa	1000: baaa	Decoding <u>0101</u> 001101:
0001: a	1001: ba	
0010: ba	1010: ca	Decoding 010 <u>1001</u> 101:
0011: aa	1011: baa	
0100: c	1100: bb	Decoding 010100 <u>1101</u> :
0101: aaa	1101: c	
0110: d	1110: d	
0111: b	1111: b	

a

a

a

*

*

*

*

...

a

a

a

b

a

*

*

...

a

a

a

b

a

c

*

...

(a) Dictionary \mathcal{W} with $K = 3$ and $O = 1$

(b) Decoding of message 101001101

(a) Dictionary \mathcal{W} with $K = 3$ and $O = 1$

(b) Decoding of message 101001101

Figure 1: Decoding example for the message 101001101 using the dictionary \mathcal{W} . Codewords of \mathcal{W} are encoded in 4 bits, but only 3 bits are consumed from the compressed source at each step ($K = 3$), and there is 1 bit of overlap between consecutive codewords ($O = 1$). The grayed bit corresponds to the initialization which in our current algorithm is always zero.

Because of the overlap, not all possible words of the dictionary are accessible at each step. This is due to the prefix which is fixed by the previous codeword. We named *chapters* to each set of accessible words. On the example represented in Fig. 1, we have two chapters. The first contains the codewords from 0000 to 0111, and the second chapter contains the codewords from 1000 to 1111. There are not repeated words within a chapter, but the same word can appear in different chapters.

VFs codes, including Marlin, have the limitation that the number of words in the dictionary $|\mathcal{W}|$ must be larger than the number of symbols of the alphabet $|A|$. Marlin requisites are more strict due to the overlapping, and 2^K must be larger than $|A|$ to achieve compression. Hence, in previous publications, we have suggested to use a value 12 for K when encoding 8-bit alphabets, forcing us upon dictionaries that contain thousands of words. This limit is the motivation for the Rice-Marlin codecs introduced in this work, which allow to reduce the size of the working alphabet and, therefore, to use a value of 8 for K .

3 Increasing the Efficiency of Marlin Dictionaries

The coding efficiency of stateless VF codecs is related to the ratio between the size of the dictionary and the size of the alphabet. However, increasing the size of the dictionary requires more memory and reduces the speed of the codec. Therefore, in this section we present two techniques that allow us to increase the efficiency of VF codecs by reducing the size of the alphabet instead of increasing the size of the dictionary. The first technique, *Removing High Entropy Bits*, is more effective on high entropy sources, while the second technique, *Removing Low Entropy Symbols*, is more effective on low entropy sources.

3.1 Removing High Entropy Bits

We analyze source symbols as a group of bits. Often, when encoding high entropy sources, the least significant order bits of each symbol can not be compressed (*i.e.*, their probability of being 0 or 1 is close to 50% and uncorrelated with other bits). This is a well known effect, and it is taken advantage from in, *e.g.*, Rice [17] codecs, where the least significant order bits are simply stored in a truncated binary form.

In VF codecs, each non-compressible bit effectively duplicates the number of alphabet symbols, hence, by encoding them in a separate section, we free a large number of codewords in the dictionary.

We use a simple variation of the Rice-Golomb coding as we split each input symbol in two parts: the S least significant bits are the *remainder*, while the most significant bits are the *quotient*, defined as:

$$r = x \bmod 2^S, \text{ and } q = \left\lfloor \frac{x}{2^S} \right\rfloor, \quad (1)$$

where x is the original symbol interpreted as an unsigned integer.

For each input message, we compress all *quotients* together using the Marlin codec, while the remainders are simply packed together afterwards. Hence, for $S = 0$, our codec is simply equivalent to the original Marlin code.

To find the optimal S for an alphabet, we build a dictionary for each possible value of S and choose the one with the best efficiency.

3.2 Removing Low Entropy Symbols

While building a VF codec, we must ensure that each possible source symbol can be represented. As a consequence, symbols that are unlikely to be generated by the source must use at least one entry in the dictionary. When using very small dictionaries together with low entropy sources, a significant portion of the entries in the dictionary are wasted representing symbols whose probability of appearing in a message is close to zero.

To solve this problem, we identify symbols whose occurrence probability is below a certain threshold, and we exclude them from the Marlin dictionary, labeling them as *unrepresented symbols*. When encoding a message, each *unrepresented symbol* is stored uncompressed in a dedicated section.

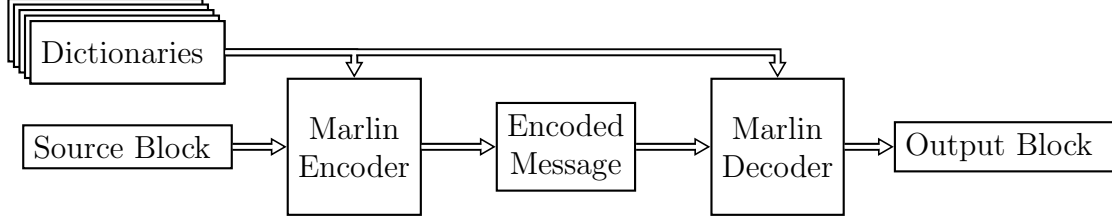


Figure 2: To improve coding speed, a set of dictionaries is built beforehand and known to both encoder and decoder. The encoder selects the best fitting dictionary, compresses the block, and emits the compressed message which includes the index of the dictionary used. The decoder recovers the original data using the selected dictionary.

4 Proposed Code Format

Most entropy encoders build custom encoding and decoding tables based on the statistics of the symbols appearing in the block being currently encoded. However, building such tables is time consuming, and several compression formats default to a predefined generic table, as in JPEG or ZIP. Marlin is optimized for speed, but we also want it to provide a competitive compression ratio, hence, we use a hybrid approach where we build beforehand a set of dictionaries that cover a wide range of possible probability distributions that may appear in the source, and we make this set of dictionaries available to both encoder and decoder, as seen in Fig. 2. Then, when compressing a block, we select the best fitting dictionary.

Also, as in previous Marlin codes, we rely on the software used to transmit the message from source to target to provide the decoder with the size of the encoded message and the size of the original message.

Our suggested message format has the following sections:



1. *#D*: 1 byte containing the index of the dictionary used to encode the message.
2. *#U*: 1 byte containing the number of unrepresented symbols in the message.
3. *quotients*: variable sized field encoding the quotients using a Marlin dictionary.
4. *unrepresented*: each unrepresented symbol encoded as a pair $\{location, symbol\}$.
5. *reminders*: bit field with all concatenated reminders.

The *quotients* section is made of concatenated Marlin words, whose size depends on the chosen dictionary. The *unrepresented* section is made by pairs $\{location, symbol\}$, those encode the symbols that the selected Marlin dictionary can not represent. In most cases this section is small or even empty. The *location* field contains the absolute position of the *symbol* within the uncompressed message, and its size depends precisely on the size of the uncompressed message: 1 byte for messages smaller than 2^8 symbols, 2 bytes for message smaller than 2^{16} symbols, 4 bytes for messages smaller than 2^{32} symbols, etc. If the original message has more than 255 *unrepresented* symbols we store the original message uncompressed.

The *reminders* section is made of the concatenated least significant bits from the original message, as seen in Fig. 3. Those can be encoded and decoded fast using bit shuffling instructions.

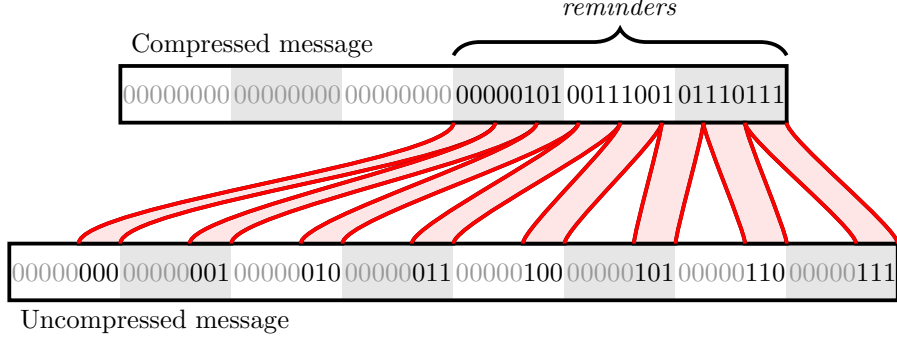


Figure 3: Reminders are stored concatenated all together in its own section. Both encoding and decoding of reminders can be done very efficiently using bit manipulation instructions.

5 Improved Encoding Algorithm

The process of encoding of a Marlin message is divided in three stages. On the first stage, we calculate the quotients for each symbol and we encode them using the VF dictionary. In this step, symbols whose quotient is not represented in the VF alphabet are replaced by the most common quotient acting as a placeholder. The second stage is to emit the location and value of each *unrepresented* symbol identified in the previous step. The third stage is to emit the *reminders* of each source symbol.

The most time consuming stage is, by far, the encoding of the VF words. The goal of this stage is to find the largest prefix of the source that corresponds to a word in the dictionary. The two common approaches for solve this problem (hash tables or prefix trees) require one query to the data structure per source symbol; as this query can not be predicted, it often causes a cache-miss, becoming the main bottleneck. Hence, to speed up encoding times we must have better locality to improve the cache hits/miss ratio. We found out that prefix trees enable us to better control locality.

We build our prefix tree in the form of a matrix organized as shown in Fig. 4. Each column of the encoding matrix represents the current state, and each row represents the next source symbol to be encoded. Each codeword in the dictionary corresponds to one state, and each cell contains both the next state, and a flag that indicates if the current state/codeword must be emitted in the compressing stream. For example, if we have encoded "aa" and thus we are on the fourth column, and we fetch an "a" from the source, we must fetch the first row of the fourth column, which would indicate to go to the state belonging to the "aaa", and not emit a word. On the other hand, if we are in "aa" and receive a "b", we fetch the second row of the fourth column, and thus we will emit the codeword corresponding to "aa" (*i.e.*, the column index), and go to the state "b" of the corresponding chapter.

This data structure has three particularities: First, we avoid one memory fetch by identifying codeword with state. Second, we avoid one memory fetch by storing the next state and the flag in the same cell. Third, by choosing this disposition of rows and columns, and sorting inputs symbols from most probable to least probable, we will create an inbalance where most memory fetches will be close together and centered on the first rows of the structure, as seen in Fig. 4, improving locality.

	aaaa	a	ba	aa	c	aaa	d	b	baaa	ba	ca	baa	bb	c	d	b
a	a!	aa	a!	aaa	a!	aaaa	a!	ba	a!	baa	a!	baaa	a!	ca	a!	ba
b	b!	b!	b!	b!	b!	b!	b!	b!	b!	b!	b!	b!	b!	b!	b!	bb
c	c!	c!	c!	c!	c!	c!	c!	c!	c!	c!	c!	c!	c!	c!	c!	c!
d	d!	d!	d!	d!	d!	d!	d!	d!	d!	d!	d!	d!	d!	d!	d!	d!

Figure 4: VF encoding table for the dictionary shown in Fig. 1. Our current state is encoded in a column, and corresponds to the current codeword. The next possible symbols are encoded as rows. Each cell contains our next state (our next codeword), as well as a flag (!) indicating that the current codeword should be emitted. Shading color indicates access probability. Red and blue symbols correspond to the two chapters of the dictionary.

6 Evaluation

We integrated our new contributions in Marlin’s codebase, which is publicly available¹. Evaluation is performed on a i5-6600K CPU at 3.5GHz with 64GB of DDR4-2133 RAM running Ubuntu 16.04 and compiled using GCC v5.4.0 with the `-O3` flag. While previous Marlin codes suggested to use a $K = 12$, our new contributions allow us to use a $K = 8$, avoiding shift operations in the decoding loop. We use $O = 4$ as the overlap parameter. In this evaluation we decorate Marlin codes with the notation (K, O) to indicate the used values for K and O . Our main evaluation metric is the *compression efficiency* defined as the ratio between the information entropy of the source and the average bit rate achieved: $\eta_X = H(X)/\text{ABR}(X)$.

6.1 Results on synthetic data

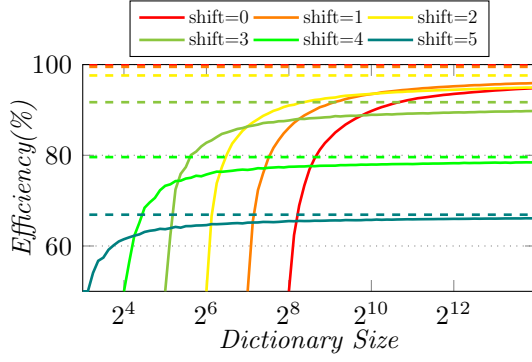
We have evaluated our new contributions on synthetic data distributions at different entropies. In Fig. 5a we show how using larger shift (S) values in our Rice-Marlin distributions allow us to achieve compression gains with smaller dictionary sizes, but also reduces the maximum efficiency achievable, as the stored reminders are not compressed at all. In Fig. 5b we observe how we must find the right shift value to use for each entropy level, with high entropy levels requiring larger shifts.

In Fig. 5c and Fig. 5d we have analyzed the impact of our high and low entropy contributions on different probability distributions, showing that both are complementary, and together we achieve compression efficiencies well above the 95% mark in almost the entire range of entropies and tested distributions.

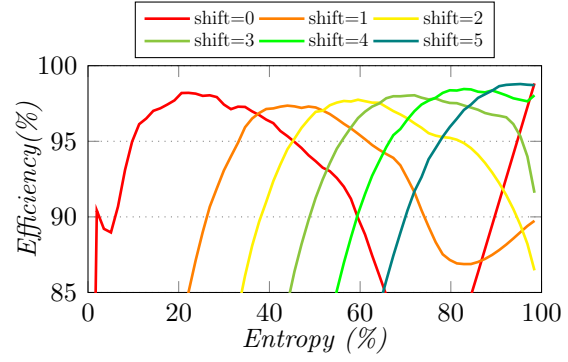
The Rice codec, being a specialized codec, is very efficient for Laplacian distributions but struggles on significantly different distributions, such as Poisson. On the other hand, Marlin is a generic VF codec that can efficiently compress sources from any probability distribution, even when used in combination with Rice encoding.

Finally, in Fig. 5e and Fig. 5f, we show our improvement against our previous versions of Marlin, which use dictionary sizes 16 times larger. There we achieve approximately the same accuracy in low entropy sources, as the large dictionaries used previously were not very sensitive to the few codewords wasted on rare symbols. However, we observe a significant improvement on high entropy sources.

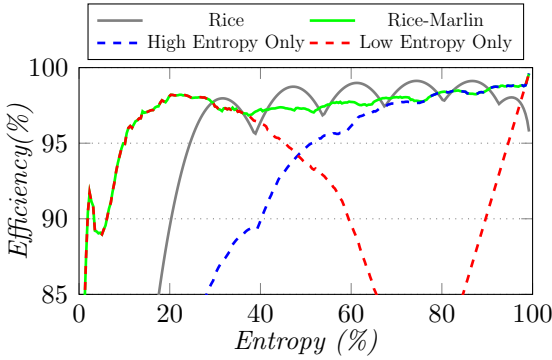
¹Git repository: <https://github.com/MartinezTorres/marlin> (tag:dcc2019)



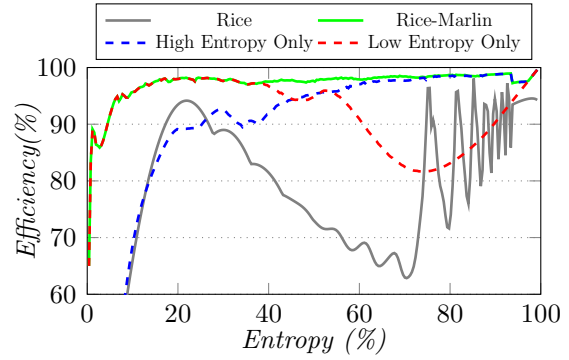
(a) Efficiency vs. Size (Laplacian, $H = 50\%$)



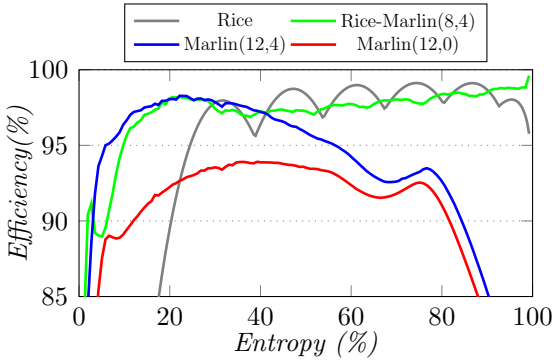
(b) Efficiency vs. Entropy (Laplacian)



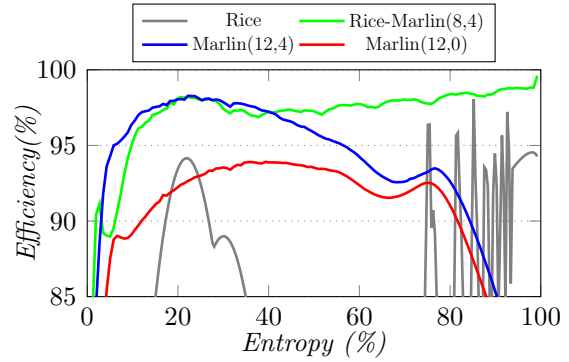
(c) Efficiency vs. Entropy (Laplacian)



(d) Efficiency vs. Entropy (Poisson)



(e) Efficiency vs. Entropy (Laplacian)



(f) Efficiency vs. Entropy (Poisson)

Figure 5: (a): larger shift values enable smaller dictionary sizes, but reduce the maximum possible efficiency achievable, which we represent in a dashed line. (b): as a consequence, we must find the right balance for each entropy level. (c) and (d): removing low entropy symbols is complementary to removing high entropy bits, and combining both provides excellent efficiency over the entire entropy range. Unlike Rice codec, our Rice-Marlin codec is generic and performs well even in non Laplacian/Exponential distributions. (e) and (f): compared to previous versions of Marlin, Rice-Marlin achieves approximately the same compression efficiency for low entropy sources while using 16x smaller dictionaries, while having significantly better efficiency for high entropy sources.

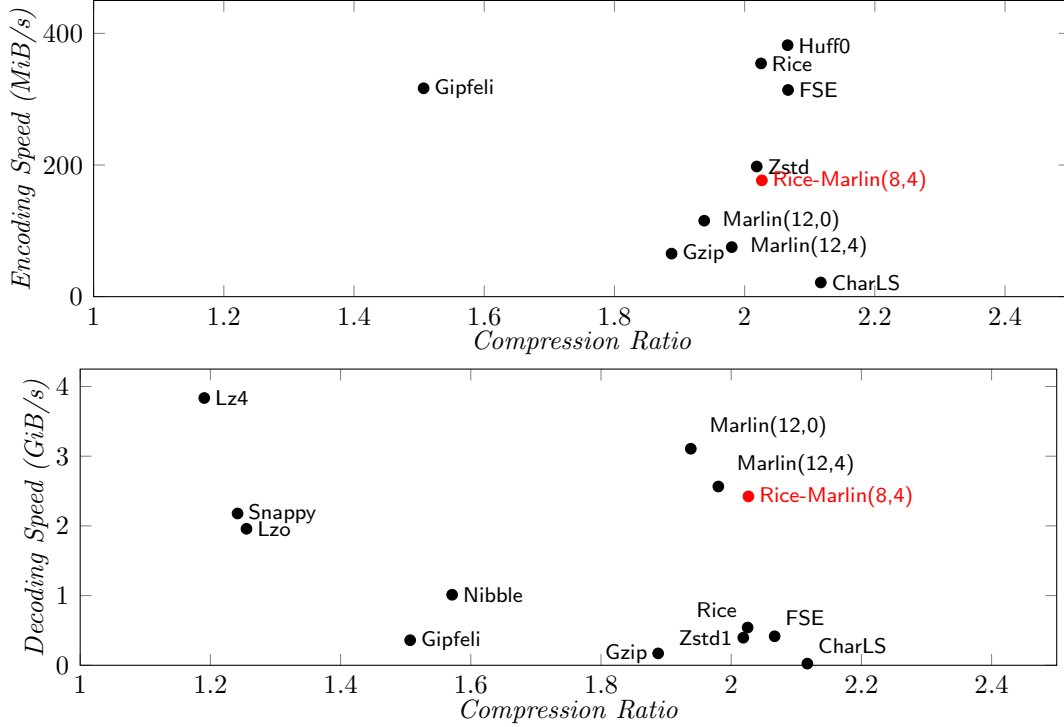


Figure 6: Evaluation on the Rawzor lossless image compression dataset [26]. Compared to Zstd, a very competitive fast compression algorithm, Rice-Marlin achieves comparable compression performance and encoding speed, while decoding 6.12x faster.

6.2 Results on real data

In Fig. 6 we test Rice-Marlin against several state-of-the-art codecs on the Rawzor [26] image set compressing independently blocks of 64×64 pixels. The prediction model uses the pixel above, and we compress the residuals.

Despite using dictionaries 16 times smaller, Rice-Marlin achieves a compression ratio of 2.02647, a 2.34% improvement over Marlin with Overlapping codes [2], and a 4.56% improvement over original Marlin [1]. Despite the increased complexity of the new decoding algorithm, the decoding speed only drops by 5.56%. On the other hand, Rice-Marlin is 2.4x faster to encode, achieving 176.67 MiB/s, mainly thanks to the smaller dictionary, and the optimizations described in section 5.

7 Conclusions

We have presented two techniques for Marlin that improve the ratio between the dictionary size and the alphabet size by reducing the alphabet size. Using those we achieve better compression efficiency while using dictionaries 16 times smaller than before. We presented an encoding algorithm that leverages the smaller dictionaries to achieve 2.4x times faster encoding times. Thanks to this improvements, our new codec, named Rice-Marlin, achieves encoding speeds and compression ratios similar to the Zstd in lossless image compression, while still being 6.12x faster to decode.

References

- [1] M. Martinez, M. Haurilet, R. Stiefelham, and J. Serra-Sagristà, “Marlin: A high throughput variable-to-fixed codec using plurally parsable dictionaries,” in *Proceedings of Data Compression Conference*. IEEE, 2017.
- [2] M. Martinez, K. Sandfort, D. Dubé, and J. Serra-Sagristà, “Improving marlin’s compression ratio with partially overlapping codewords,” in *Proceedings of Data Compression Conference*. IEEE, 2018.
- [3] Y. Collet, “LZ4,” [lz4.github.io/lz4](https://lzh4.github.io/lz4/), 2011.
- [4] M. Oberhumer, “LZO: Lempel Zip Oberhumer,” www.oberhumer.com/opensource/lzo, 1996.
- [5] Z. Tarantov and S. Gunderson, “Snappy,” google.github.io/snappy, 2011.
- [6] P. Katz, “Deflate lossless data compression,” US patent 5051745, 1991.
- [7] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” *IEEE Transactions on Information Theory*, vol. 23, no. 3, pp. 337–343, 1977.
- [8] D. A. Huffman *et al.*, “A method for the construction of minimum-redundancy codes,” *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952.
- [9] J. Rissanen, “Generalized Kraft inequality and arithmetic coding,” *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, 1976.
- [10] G. N. N. Martin, “Range encoding: an algorithm for removing redundancy from a digitised message,” in *Proc. Institution of Electronic and Radio Engineers International Conference on Video and Data Recording*, 1979.
- [11] R. Lenhardt and J. Alakuijala, “Gipfeli-high speed compression algorithm,” in *Proceedings of Data Compression Conference*. IEEE, 2012, pp. 109–118.
- [12] R. N. Williams, “An extremely fast Ziv-Lempel data compression algorithm,” in *Proceedings of Data Compression Conference*. IEEE, 1991, pp. 362–371.
- [13] D. Harnik, E. Khaitzin, D. Sotnikov, and S. Taharlev, “A fast implementation of Deflate,” in *Proceedings of Data Compression Conference*. IEEE, 2014, pp. 223–232.
- [14] Y. Collet, “Huff0,” fastcompression.blogspot.de/p/huff0-range0-entropy-coders.html, 2013.
- [15] —, “Finitestateentropy,” github.com/Cyan4973/FiniteStateEntropy, 2013.
- [16] J. Duda, “Asymmetric numeral systems,” *CoRR*, vol. abs/0902.0271, 2009. [Online]. Available: <http://arxiv.org/abs/0902.0271>
- [17] R. Rice and J. Plaunt, “Adaptive variable-length coding for efficient compression of spacecraft television data,” *IEEE Transactions on Communication Technology*, vol. 19, no. 6, pp. 889–897, 1971.
- [18] B. Greenwood, “Lagarith,” lags.leetcode.net/codecs.html, 2011.
- [19] J. de Vaan, “CharLS,” github.com/team-charls/charls, 2007.
- [20] B. P. Tunstall, “Synthesis of noiseless compression codes,” *Ph.D. dissertation, Georgia Institute of Technology*, 1967.
- [21] S. A. Savari, “Variable-to-fixed length codes and plurally parsable dictionaries,” in *Proceedings of Data Compression Conference*. IEEE, 1999, pp. 453–462.
- [22] A. Al-Rababa’a and D. Dubé, “Using bit recycling to reduce the redundancy in plurally parsable dictionaries,” in *14th Canadian Workshop on Information Theory*. IEEE, 2015, pp. 62–65.
- [23] S. Yoshida and T. Kida, “An efficient algorithm for almost instantaneous VF code using multiplexed parse tree,” in *Proceedings of Data Compression Conference*. IEEE, 2010, pp. 219–228.
- [24] H. Yamamoto and H. Yokoo, “Average-sense optimality and competitive optimality for almost instantaneous VF codes,” *IEEE Transactions on Information Theory*, vol. 47, no. 6, pp. 2174–2184, 2001.
- [25] D. Dubé and F. Haddad, “Individually optimal single- and multiple-tree almost instantaneous variable-to-fixed codes,” in *2018 IEEE International Symposium on Information Theory*, 2018, pp. 2192–2196.
- [26] S. Garg, “The new test images,” www.imagecompression.info/test_images, 2011.