Marlin: A High Throughput Variable-to-Fixed Codec using Plurally Parsable Dictionaries

Manuel Martinez*, Monica Haurilet*, Rainer Stiefelhagen* and Joan Serra-Sagristà†

*Karlsruhe Institute of Technology
Karlsruhe, 76131, Germany
name.surname@kit.edu

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

Abstract

We present Marlin, a variable-to-fixed (VF) codec optimized for decoding speed. Marlin builds upon a novel way of constructing VF dictionaries that maximizes efficiency for a given dictionary size. On a lossless image coding experiment, Marlin achieves a compression ratio of 1.94 at 2494MiB/s. Marlin is as fast as state-of-the-art high-throughput codecs (*e.g.*, Snappy, 1.24 at 2643MiB/s), and its compression ratio is close to the best entropy codecs (*e.g.*, FiniteStateEntropy, 2.06 at 523MiB/s). Therefore, Marlin enables efficient and high-throughput encoding for memoryless sources, which was not possible until now.

Introduction

The goal of High Throughput (HT) compression is not to reduce storage requirements, but to better leverage the available bandwidth. In this scenario, the maximum throughput is a balance of compression ratio and coding/decoding speed. A number of algorithms exploit different sweet spots, depending on the required application.

All current HT algorithms are derived from LZ77 [1] using hash tables [2, 3] for encoding, and memory copies for decoding, *e.g.*, Snappy [4], LZ4 [5], or LZO [6]. The entropy stage is either dropped, *e.g.*, Snappy, LZ4, or greatly simplified, *e.g.*, LZO, Gipfeli [7], since adaptive entropy codes like Huffman [8] and arithmetic [9]/range encoding [10] are generally slow, despite of the current efforts to make them faster, *e.g.*, Huff0 [11] for Huffman and FiniteStateEntropy [12, 13] for range encoding.

Dropping or simplifying the entropy coding stage is a reasonable trade-off for text based sources, but has a steep penalty mainly when compressing memoryless sources like noisy sensor data, *e.g.*, raw images [14, 15, 16]. To enable HT compression in such cases, we present Marlin, an entropy codec that can be decoded as fast as Snappy and LZO. Marlin follows a variable-to-fixed (VF) coding approach.

Variable-to-fixed codes are similar in essence to LZ77 and can be decoded equally fast. In VF coding, a memoryless digital source is described as a sequence of variable sized words from a given dictionary. The code emitted consists of the indexes that point to such words. The dictionaries are generally built to be uniquely parsable, which requires that no word is a prefix of any other word in the dictionary. Such dictionaries can be build using Tunstall [17] algorithm, but they tend to be large.

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 2014SGR-691. This work is also partially supported by the German Federal Ministry of Education and Research (BMBF) within the SPHERE project.

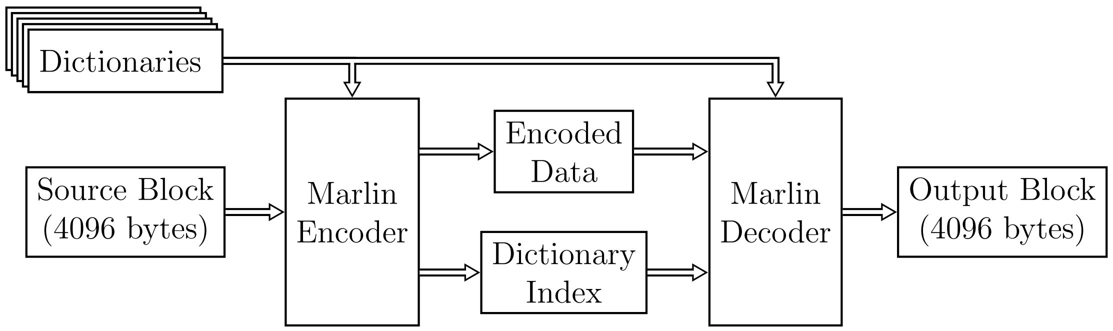


Figure 1: Marlin encodes blocks of 4096 bytes. A dictionary set is known to both encoder and decoder. The encoder inspects the block, selects the best dictionary, compresses the block, and emits the compressed blob and the index of the dictionary. The decoder recovers the original block using the selected dictionary.

Plurally *–i.e., not uniquely–* parsable dictionaries [18] have been shown to outperform Tunstall dictionaries of equal size [19, 20, 21]. We suggest a novel way of constructing plurally parsable dictionaries for arbitrary alphabets using a constrained Markov process of only 255 states (for 8-bit sources). To optimize the dictionary to an input source, we apply an iterative algorithm with two steps: the first step calculates the Markov process for a given dictionary, the second step creates an optimized dictionary for a given Markov process. Our algorithm alternates between the two steps. Our proposed HT encoding/decoding technique, Marlin[1], builds upon this dictionary to implement an algorithm that is extremely fast to decode.

In order to assess the coding performance of Marlin, we evaluate it for synthetic distributions, achieving better compression efficiency than Tunstall over all entropy levels when limited to 2^{12} entries. To evaluate Marlin in a real application, we test it on the well known Rawzor image compression benchmark [22]. Marlin achieves a compression ratio of 1.94 while decoding at 2494 MB/s, and Tunstall only achieves 1.34 while decoding at 1815 MB/s. As a reference, JPEG-LS [15], achieves on the same benchmark a compression ratio of 2.12 and a decoding speed of 30.6 MB/s.

Proposed Encoding/Decoding Technique

High throughput codes are rarely used to compress files, therefore Marlin, like LZ4 [5], Snappy [4], Huff0 [11] and FSE [12], does not define a file format. Marlin is designed to compress data blocks of a fixed size (see Fig. 1). Each block is encoded using a dictionary chosen from a known set of dictionaries. This set should cover all expected probability distributions that the source may present. The encoder generates a compressed data blob, and provides the index of the dictionary used to compress the block. The decoder, which knows the same dictionaries used for compression, receives the compressed data blob and the dictionary index, and generates the original uncompressed data.

[1]We choose the name for the Black Marlin, the fastest fish known to man. Like fish, which are not known for having great memory, our algorithm does not remember the state of the input.

Implementation Details

Block size. Small block sizes provide better compression efficiency, but large blocks improve decompression performance. We fix the block size to 4096 bytes, which corresponds to the size of a memory page in x86 processors.
Dictionary set. The dictionary building process is time consuming, and the generated dictionaries are too large to be sent with the data. This makes it unfeasible to create a custom dictionary per block (as Huff0 and FSE do). Marlin uses a fixed set of dictionaries with common probability distributions, and selects the best fitting one to compress each block. For efficiency reasons, the dictionary set must be small, as switching dictionaries trashes the cache. We employ 11 dictionaries per distribution.
Alphabet size. Marlin currently supports 8-bit alphabets.
Dictionary size. Marlin dictionaries can contain between 2^9 and 2^{16} entries. Larger dictionaries provide better compression efficiency, but are slower to decode. We use 2^{12} entries per dictionary as it fits within most L1 caches.
Special cases. Like Huff0 and FSE, Marlin implements two special cases. One for blocks that contain only zeros, and one for blocks which can not be compressed at all. Both cases are indicated by magic values in the dictionary index.

Encoding

The first step of the encoding process is to select a dictionary to encode the current block. A naive way to find the most efficient dictionary is to encode the block with all available dictionaries, and choose the one that generates the smaller blob, but this is very inefficient. Instead, we use the block entropy to select which dictionary to use.

Given a dictionary, the encoder needs to find the largest word in the dictionary that matches the input source. We have implemented this task efficiently using a trie (*i.e.*, also known as prefix tree). The performance is bound by memory latency, therefore using smaller dictionaries boost encoding speed. This happens because a larger proportion of the trie fits within the cache, increasing the hit ratio. The compression speed of our current implementation ranges between 100MB/s and 300MB/s, however there are still many tradeoffs to be explored in this area.

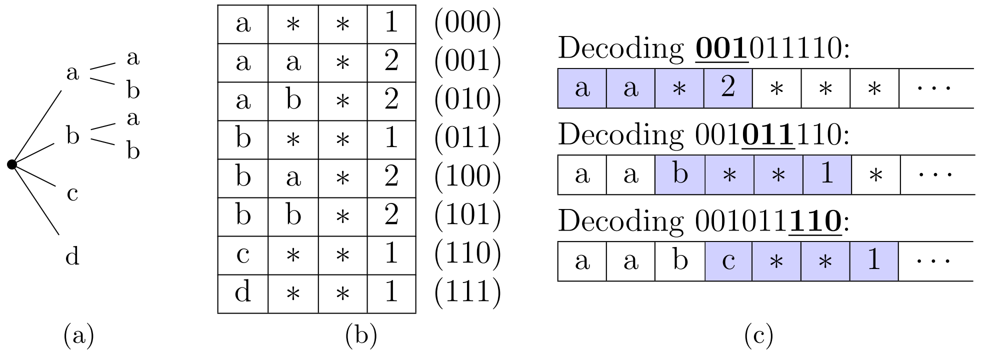
Decoding

The decoder receives a compressed blob representing a single data block, and the index of the dictionary used to compress it.

For each dictionary, the decoder prepares a table structure that contains a word per entry (see Fig. 2.b). Each entry from the table has the same length, which is a power of two. The word is at the beginning of the entry, and the word length is placed on the last byte.

The decoder loop does:

1. extract a word index from the input stream,
2. fetch the corresponding entry from the decoding table,
3. copy the entire entry to the output stream,
4. increase the output pointer accordingly.



a	∗	∗	1	(000)
a	a	∗	2	(001)
a	b	∗	2	(010)
b	∗	∗	1	(011)
b	a	∗	2	(100)
b	b	∗	2	(101)
c	∗	∗	1	(110)
d	∗	∗	1	(111)

a	a	∗	2	∗	∗	∗	⋯
a	a	b	∗	∗	1	∗	⋯
a	a	b	c	∗	∗	1	⋯

Figure 2: Decoding example where *aabc* is coded with the dictionary (a) to generate *001-011-110*. The decoder generates the table (b) from (a), storing the word length in the last byte. The decoding process (c) involves copying the whole entry to the output stream, but advance the pointer only the actual word length.

We analyze now in deeper detail the decoding process. In the first step we extract a word index from the input stream, as we are coding 8-bit streams, our dictionary needs to have more than 2^8 entries to achieve any compression at all. Actually, the larger the dictionary is, the better compression will be achieved, but we found empirically that there is little to gain for dictionaries larger than 2^{16} entries. Constraining the dictionary sizes to power of two values ensures that we can extract indexes from the input stream using only bit mangling operations. In our experiments we select 2^{12}-sized dictionaries since extracting 2 indexes from 3 bytes is efficient.

For each index, we fetch its corresponding entry from the decoding table. The whole entry is fetched from memory, even if the actual word is smaller than the maximum entry size. To avoid a performance penalty, we ensure that the table entries are aligned to the size of a cache line (64 bytes, usually).

Next, we leverage the recently added ability of current processors of performing unaligned writes with almost no penalty, by copying the entire entry to the output stream. Again, copying the whole entry instead of only the relevant symbols of the word is performed to avoid checks. Finally, we increase the output pointer by the value stored in the last byte of the entry.

This algorithm has two key advantages. First, it has a deterministic input pipeline allowing parallel memory fetches; secondly, both word and word length are retrieved from a single memory fetch.

Dictionary generation

We define $A = \{a_1, a_2, \cdots, a_N\}$ as an alphabet of input symbols sorted in order of non-increasing probability ($i.e.$, $P(a_n) \geq P(a_{n+1}), \forall n$). We generate a dictionary $\mathcal{W}$ in the form of a tree where each node corresponds to an input symbol.

We build $\mathcal{W}$ as follows: First, we initialize the tree with a leaf for each input symbol. Then, while $|\mathcal{W}|$ is smaller than desired, we add a single child to the most probable node. The new node will contain the symbol a_{i+1} where i is the current number of leaves of the parent node. As a special case, if a node has only one remaining leaf to grow, this leaf is grown automatically. See Fig. 3 for an example.

Calculating word probabilities for a given Markov process

As we generate plurally parsable dictionaries, the compression process is not steady and we model it as a Markov process with $N-1$ possible states $S = s_1, s_2, \cdots, s_{N-1}$. On s_i, words can not start with symbols with an index smaller than i.

We define the probability of the word w given that we are in state s_i as:

$$P(w|s_i) = \frac{P(w_1)}{\Sigma_{n=i}^{N} P(a_n)} \cdot \prod_{n=2}^{|w|} P(w_n) \cdot \sum_{n=1+c(w)}^{N} P(a_n), \tag{1}$$

where w_n is the n'th symbol of w, and $c(w)$ is the number of children of the last node of w. Thus the non-conditioned probability of w is:

$$P(w) = \sum_{i}^{N-1} P(s_i) \cdot P(w|s_i), \tag{2}$$

where $P(s_i)$ is the steady probability of being in state s_i. Therefore, we need to know the steady probabilities of the Markov process to calculate word probabilities.

Calculating the steady state probabilities of a Markov process for a given dictionary

We calculate now the transition matrix for the Markov process T:

$$T_{ij} = \sum_{w \in \mathcal{W}_j} P(w|s_i), \tag{3}$$

where T_{ij} is transition probability from s_i to s_j, and $\mathcal{W}_j$ are the words that end at s_j.

Then, the steady probabilities of being in each state $P(s_i)$ correspond to the first row of T^n for $n \to \infty$ (we must note that T^n converges after only a few iterations).

Building the dictionary

We initialize the dictionary building process with a trivial Markov process corresponding to a unique parsable dictionary (see Fig. 3.a). Then we build a full dictionary as previously stated (Fig. 3.e). We use this dictionary to estimate a new Markov process (Fig. 4.a), and build a new dictionary (Fig. 4.e).

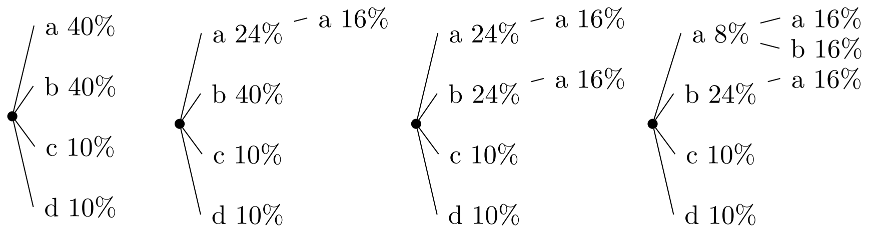
This process is repeated until convergence (Fig. 5.b), which we found empirically to happen in a few iterations.

The average bit rate of the dictionary in bits is:

$$\text{ABR}(\mathcal{W}) = \frac{log_2(|\mathcal{W}|)}{\sum\limits_{w \in \mathcal{W}} P(w) \cdot |w|}. \tag{4}$$

$\hat{P}_1(S_i)$

S_a	100%
S_b	0%
S_c	0%
S_d	0%

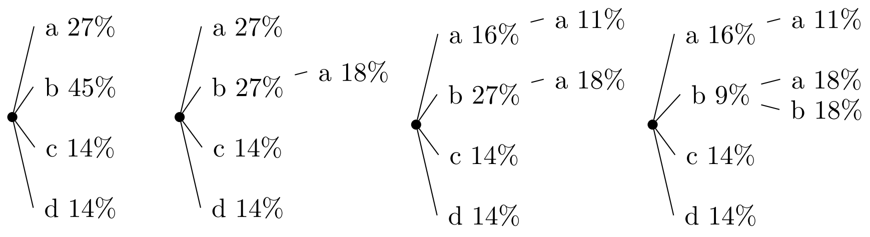


(a) Process (b) Initial (c) Step 1 (d) Step 2 (e) Step 3

Figure 3: $A = \{a, b, c, d\}$ with probabilities $\{40\%, 40\%, 10\%, 10\%\}$, (a) naive initialization of the Markov process steady probabilities, $\hat{P}_1(S_i)$, (b) tree initialization, (c)-(e) growing steps.

$\hat{P}_2(S_i)$

S_a	68%
S_b	27%
S_c	5%
S_d	0%

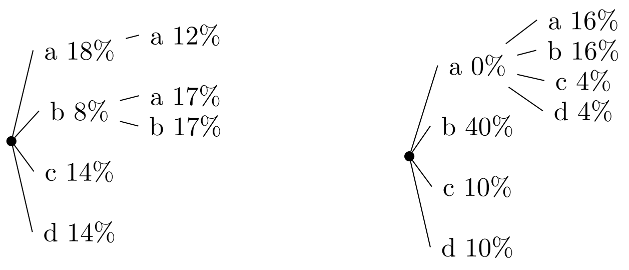


(a) Process (b) Initial (c) Step 1 (d) Step 2 (e) Step 3

Figure 4: Second stage of the dictionary creation. (a) second stage steady probabilities of the Markov process, $\hat{P}_2(S_i)$, which are estimated from the dictionary obtained in the previous stage (Fig. 3.e), (b)-(e) the tree is grown again.

$\hat{P}_3(S_i)$

S_a	74%
S_b	18%
S_c	8%
S_d	0%



(a) Process (b) Marlin $\text{ABR}(\mathcal{W}) = 1.45$ (c) Tunstall $\text{ABR}(\mathcal{W}) = 1.4$

Figure 5: Third stage of the dictionary creation. (a) $\hat{P}_3(S_i)$ are estimated from the previous stage dictionary (Fig. 4.e), (b) the resulting tree is identical to Fig. 4.e, no more iterations are needed, (c) equivalent Tunstall dictionary, which is less efficient.

Evaluation

We compare Marlin to the following algorithms:

Tunstall [17]: we use Marlin code with Tunstall dictionaries.

Rice [23]: our implementation of Rice-Golomb codes (using the Rice subset) which is a popular entropy codec for Laplace and Exponential distributions [24].

Nibble [25]: used by Kinect and the fastest entropy encoder we are aware of. Values in $[-7, 7]$ are encoded in a nibble (4-bit word), larger values are emitted raw.

Snappy [4]: Google's high throughput codec based on LZ77 without entropy stage.

LZO [6]: first high throughput algorithm that saw widespread use. It is used in the linux kernel and the btrfs file system. We evaluate the Lzo1-15 variety.

LZ4 [5]: state-of-the-art high throughput algorithm developed by Yann Collet. Its current implementation is faster than Snappy due to improved hashing [26].

Gipfeli [7]: algorithm developed from Snappy that adds a simple entropy stage.

FiniteStateEntropy [12]: state-of-the-art implementation by Yann Collet of Jarek Duda's Asymmetric Numerical Systems theory [13]. It is related to range encoding, but requires only one value to keep the state, making it faster.

Zstandard [27]: a compression algorithm developed by Yann Collet that builds upon LZ4 and FiniteStateEntropy. We evaluate its fastest setting.

Gzip [28]: is often used as the reference to evaluate high throughout algorithms. It uses the deflate algorithm (LZ77 + Huffman).

CharLS [16]: fast JPEG-LS implementation for lossless image compression.

All our implementations are coded in C++ and publicly available[2]. Great effort has been taken to implement all algorithms as efficiently as possible, however no assembly code was employed to keep the code portable. Evaluation is performed on an i5-6600K CPU at 3.5GHz with 64GB of DDR4-2133 running Ubuntu 16.04 and GCC v5.4.0 with the -O3 flag.

Synthetic results. We evaluate the performance of Marlin on a circular Laplace distribution. This distribution arises often when dealing with differential data [29]. We report compression efficiency, encoding and decoding speed at entropy levels from 1% to 99%. Being a memoryless source, its compression ratio is limited by its entropy [30], therefore the compression efficiency is the ratio between the entropy and the average bit rate achieved: $\eta_X = H(X)/\text{ABR}(X)$. Fig. 6 shows that Marlin achieves compression efficiencies well above 80% for the entire entropy range. Only FiniteStateEntropy and Rice are able to compress better, but are significantly slower.

Results on real data. A typical application for entropy codecs is to compress noisy sensor data (*e.g.*, lossless image compression). We use Marlin to compress the Rawzor [22] image set using blocks of 64×64 pixels (4096 bytes) which are processed independently. The prediction model uses the pixel above, and we compress the residuals. Fig. 7 shows that Marlin achieves decompression speeds in the same range as Snappy, while the compression ratio is on the same range as Zstd and Rice, and close to FiniteStateEntropy.

[2]https://github.com/MartinezTorres/Marlin

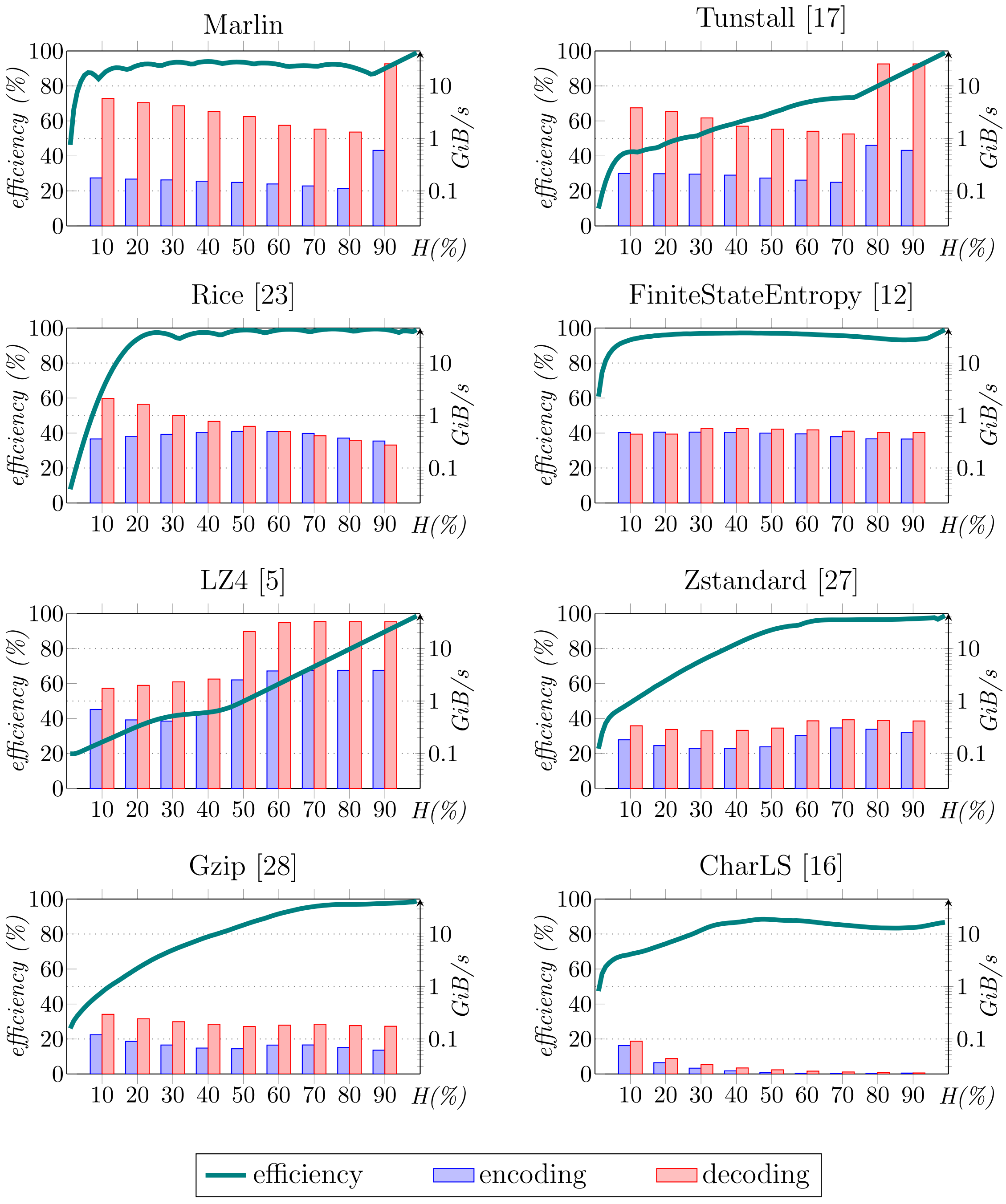


Figure 6: Evaluation on synthetic memoryless Laplace sources of varying entropies. Marlin's compression efficiency is above 80% and its decoding speed is above 1GiB/s. Tunstall, with equally sized dictionaries, is significantly less efficient. FiniteStateEntropy and Rice compress better, but are slower. LZ4 speed is comparable, but has poor efficiency. Note that speeds above 10GiB/s correspond to uncompressed data.

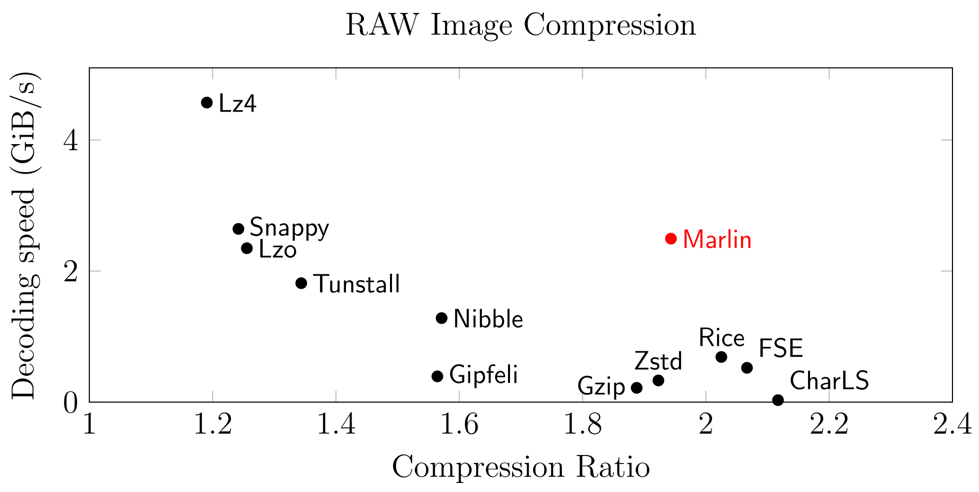


Figure 7: Evaluation on the Rawzor image compression dataset. The current state-of-the-art algorithm for high throughput entropy coding, FiniteStateEntropy (FSE) achieves a compression ratio of 2.07 at 524 MiB/s. Our algorithm, Marlin, achieves a compression ratio of 1.94 at 2494 MiB/s, which is 4.76 times faster.

Conclusions and future work

We have presented Marlin, a high-throughput compression algorithm that achieves competitive compression efficiency for memoryless sources. Marlin builds upon a novel variable-to-fixed code using plurally parsable dictionaries, and a very optimized branchless decoding algorithm. As a consequence, Marlin achieves decoding speeds above the GiB/s range. However, the current implementation is to be understood as a proof of concept, and we expect to improve the decoding rate using processor intrinsics and new optimization techniques. Likewise, as plurally parsable dictionaries enable tradeoffs between compression ratio and encoding speed, we are currently investigating the best choices.

References

[1] 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.

[2] R. N. Williams, “An extremely fast Ziv-Lempel data compression algorithm,” in *Proceedings of Data Compression Conference*. IEEE, 1991, pp. 362–371.

[3] 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.

[4] Z. Tarantov and S. Gunderson, “Snappy,” code.google.com/p/snappy, 2011, [Accessed 20-October-2016].

[5] Y. Collet, “LZ4,” lz4.org, 2011, [Accessed 20-October-2016].

[6] M. Oberhumer, “LZO: Lempel Zip Oberhumer,” www.oberhumer.com/opensource/lzo, 1996, [Accessed 20-October-2016].

[7] R. Lenhardt and J. Alakuijala, “Gipfeli-high speed compression algorithm,” in *Proceedings of Data Compression Conference*. IEEE, 2012, pp. 109–118.

[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] Y. Collet, "Huff0," fastcompression.blogspot.de/p/huff0-range0-entropy-coders.html, 2013, [Accessed 20-October-2016].

[12] ——, "Finitestateentropy," github.com/Cyan4973/FiniteStateEntropy, 2013, [Accessed 20-October-2016].

[13] J. Duda, "Asymmetric numeral systems," *CoRR*, vol. abs/0902.0271, 2009. [Online]. Available: http://arxiv.org/abs/0902.0271

[14] P. G. Howard and J. S. Vitter, "Fast and efficient lossless image compression," in *Proceedings of Data Compression Conference.* IEEE, 1993, pp. 351–360.

[15] M. J. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS," *IEEE Transactions on Image Processing*, vol. 9, no. 8, pp. 1309–1324, August 2000.

[16] J. de Vaan, "CharLS," github.com/team-charls/charls, 2007, [Accessed 20-October-2016].

[17] B. P. Tunstall, "Synthesis of noiseless compression codes," *Ph.D. dissertation, Georgia Institute of Technology*, 1967.

[18] S. A. Savari, "Variable-to-fixed length codes and plurally parsable dictionaries," in *Proceedings of Data Compression Conference.* IEEE, 1999, pp. 453–462.

[19] 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.

[20] 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.

[21] 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.

[22] S. Garg, "The new test images," www.imagecompression.info/test_images, 2011, [Accessed 20-October-2016].

[23] 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.

[24] B. Greenwood, "Lagarith," lags.leetcode.net/codec.html, 2011, [Accessed 20-October-2016].

[25] H. Martin, "Openkinect," openkinect.org, 2010, [Accessed 20-October-2016].

[26] Y. Collet, "xxhash," cyan4973.github.io/xxHash/, 2013, [Accessed 20-October-2016].

[27] ——, "Zstandard," facebook.github.io/zstd/, 2015, [Accessed 20-October-2016].

[28] J.-l. Gailly, "Gzip," www.gnu.org/software/gzip/, 1992, [Accessed 20-October-2016].

[29] J. Li, M. Gabbouj, J. Takala, and H. Chen, "Laplacian modeling of DCT coefficients for real-time encoding," in *IEEE International Conference on Multimedia and Expo*, 2008.

[30] C. E. Shannon and W. Weaver, "The mathematical theory of communication," *The University of Illinois Press*, 1949.