Improving Marlin's Compression Ratio with Partially Overlapping Codewords

Manuel Martinez*, Kai Sandfort, Danny Dubé† and Joan Serra-Sagristà‡

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

†Université Laval
Ville de Québec, Canada
Danny.Dube@ift.ulaval.ca

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

Abstract

Marlin [1] is a Variable-to-Fixed (VF) codec optimized for decoding speed. To achieve its speed, Marlin does not encode the current state of the input source, penalyzing compression ratio. In this paper we address this penalty by partially encoding the current state of the input in the lower bits of the codeword. Those bits select which *chapter* in the dictionary must be used to decode the next codeword. Each *chapter* is specialized for a subset of states, improving compression ratio. At the same time, we use one *victim chapter* to encode all rare symbols, increasing the efficiency of the rest of them. The decoding algorithm remains the same, only now codewords have overlapping bits. Mapping techniques allow us to combine common *chapters* and thus keep an efficient use of the L1 cache. We evaluate our approach with both synthetic and real data sets, and show significant improvements in low entropy sources, where compression efficiency can improve from 93.9% to 98.6%.

1 Introduction

In High Throughput (HT) compression, compression efficiency is secondary to coding speed. Current lossless HT algorithms are based on LZ77 [2], and there is active research investigating novel ways to find matches in the LZ77 dictionary that offer a compelling tradeoff between coding speed and efficiency [3, 4]. Snappy [5], LZ4 [6], and LZO [7] are the popular implementations of this trend. In HT, the entropy stage from LZ77 is either dropped, *e.g.*, as in Snappy and LZ4, or greatly simplified, *e.g.*, as employed in LZO and Gipfeli [8]. This is because entropy codecs like Huffman [9], arithmetic [10] and range encoding [11] are considered too slow for HT.

Of all entropy codec variations, VF codes have the ability of being decompressed with a branchless loop and do not require expensive bit mangling operations. For this reason, they are promising candidates for HT coding. However, there are two more factors that affect their decoding speed. First, the decoding process must be efficiently pipelined. This is achieved by banning temporal dependencies between

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 KonsensOP project.

words, *i.e.*, a codeword must always represent the same dictionary word. Second, the size of the dictionary should fit into the L1 cache of the CPU.

For optimal codecs these two requirements are mutually exclusive. For one, Tunstall [12] proposed VF dictionaries can be pipelined, but the dictionary must be large to be efficient. Opposed to this, plurally parsable dictionaries [13, 14, 15, 16] are compact, but have temporal dependencies and thus can not be pipelined.

In 2017, Marlin [1] was introduced. Marlin is a VF codec based on plurally parsable dictionaries that achieves compelling compression ratios while being faster than most HT codecs. Marlin leverages the compactness of plurally parsable dictionaries to make them fit into the L1 cache, but eschews temporal dependencies between codewords to achieve fast decoding speeds, sacrificing compression ratio.

In this paper, we address this limitation by proposing a method to partially encode the dependencies between codewords named Partially Overlapping Codewords.

Usually, the bits used to encode the codewords merely act as indexes, and are arbitrarily chosen. This makes sense as one can always store all necessary information related to a codeword in a lookup table, and the codeword just needs to point to the corresponding entry. The cost of this technique is an extra memory access, which we must avoid to achieve state-of-the-art performance.

We suggest to sort the table indexes in a way that the least significant bits of the codeword are used to communicate dependency information between consecutive words, and thus the next codeword will be retrieved from a better fitting section of the dictionary, which we name *chapter*.

By storing the dependency information in the compressed message instead of using a lookup table, we are able to decode the message using a single memory access per codeword. More importantly, the process of decoding a codeword is stateless, hence decoding one codeword does not depend on the decoding process of the previous one, and thus it can be efficiently pipelined.

By using the least significant bits of the previous codeword to carry information about the *chapter*, the decoder sees the stream as a concatenation of independent codewords, just that there are a few bits overlapping between codewords, as seen in Fig. 1, hence we name this technique Partially Overlapping Codewords. The decoding algorithm which we used in [1] can be straightforwardly modified to cope with the overlap without any speed penalty.

Our technique allows Marlin to improve its compression ratio, but storing several *chapters* makes the dictionary larger, and thus slower, than in the original Marlin. However, we keep the dictionary small by mapping identical *chapters* into the same physical memory range using the Translation Lookaside Buffer of the CPU.

We have evaluated our technique with both synthetic and real data sets, where it shows a significant improvement in compression efficiency (*i.e.*, the ratio between the Shannon entropy [17] of the source and the mean length of the compressed message). This improvement is particularly visible for low entropy sources, where we can reach a compression efficiency of 98.6% while the original Marlin can only reach 93.9%.

As a result, the overlapping codewords technique allows to improve the compression efficiency of Marlin at a small speed penalty, thus making it a better HT entropy codec.

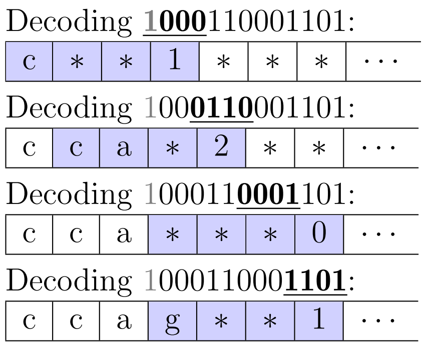
2 Proposed Code Format

The VF codec presented has two unique characteristics. To illustrate them we describe the proposed code format as well as the decoding process. As with most VF codecs, we have a dictionary of 2^N words known to both the encoder and decoder, where N is the size of the codewords in bits. A normal VF codec would consume N bits from the compressed feed, find the corresponding word in the dictionary, and emit such word. Conversely, our method *peeks* N bits from the compressed feed and emits the corresponding word, but it only consumes $N - O$ bits from the compressed feed. Here, O represents the number of *overlapping bits* between codewords. For the case $O = 0$, the format is identical to that of the original Marlin [1].

The overlap is the first unique characteristic of this codec. The second unique characteristic is that, in some cases, more than one codeword may be required to encode a single input symbol (*i.e.*, the dictionary may contain empty words). An example of this happening can be seen on the encoding of the symbol g in Fig. 1.

Chapter 0:					*Chapter 1:*				
a	a	$*$	2	(0000)	c	$*$	$*$	1	(1000)
$*$	$*$	$*$	0	(0001)	$*$	$*$	$*$	0	(1001)
a	b	$*$	2	(0010)	b	$*$	$*$	1	(1010)
b	$*$	$*$	1	(0011)	f	$*$	$*$	1	(1011)
b	a	$*$	2	(0100)	d	$*$	$*$	1	(1100)
c	$*$	$*$	1	(0101)	g	$*$	$*$	1	(1101)
c	a	$*$	2	(0110)	e	$*$	$*$	1	(1110)
a	$*$	$*$	1	(0111)	a	$*$	$*$	1	(1111)

(a) Decoding Table


Decoding **1000**110001101:
c ∗ ∗ 1 ∗ ∗ ∗ ···
Decoding 100**0110**001101:
c c a ∗ 2 ∗ ∗ ···
Decoding 100011**0001**101:
c c a ∗ ∗ ∗ 0 ···
Decoding 100011000**1101**:
c c a g ∗ ∗ 1 ···

(b) Decoding Process

Figure 1: Decoding example for the message 000110001101 using the table (a) where the word length is stored in the last byte. We use a word overlap of 1 bit, hence the dictionary is split in 2^1 chapters*. The grayed bit corresponds to the initialization, and points to* chapter 1*. At each iteration, a 4-bit codeword is read and its corresponding table entry is copied to the output stream, while the input advances 3 bits. Empty words are allowed, thus not all* chapters *are required to contain all symbols.*

3 Encoding Technique

The compression format allows for considerable freedom of how to create and encode the dictionary. We suggest an incremental technique to generate the dictionary that we have shown empirically to outperform the state of the art.

We employ the following formulation: codewords have a size of N bits, K is the number of non-overlapping bits of the codeword, and O is the number of overlapping bits, hence $K+O = N$. The dictionary, denoted by $\mathcal{W}$, has 2^N entries, and we divide it into 2^O consecutive *chapters*, $\mathcal{W}_c$ with running index c, of 2^K words each.

By this formulation, each codeword selects uniquely a chapter in which the next codeword will be decoded.

3.1 Chapter generation

To generate the chapters, we extend the algorithm used to create dictionaries in [1]. 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 chapter $\mathcal{W}_c$ in the form of a tree where each node corresponds to an input symbol.

We build $\mathcal{W}_c$ as follows: First, we initialize the tree with a leaf for each input symbol. Then, while $|\mathcal{W}_c|$ is smaller than 2^K, 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. Each node with a non-zero probability corresponds to a word in $\mathcal{W}_c$, and for practical reasons the empty word is always included, ensuring that each chapter forms an exhaustive code.

The code defined by a chapter needs not to be prefix free and, as such, it is said to be plurally parsable. As a consequence, the compression process is not steady, and the input can be left in one of $N-1$ possible states in $S = s_1, s_2, \cdots, s_{N-1}$. On s_i, words can not start with symbols with an index smaller than i.

The probability of a word w being selected given that we are in the state s_i and in the chapter $\mathcal{W}_c$ is computed by

$$P(w \mid s_i \cap \mathcal{W}_c) = \frac{1}{\Sigma_{n=i}^{N} P(a_n)} \cdot \prod_{n=1}^{|w|} P(w_n) \cdot \sum_{n=1+ch(w)}^{N} P(a_n),$$

where w_n is the n-th symbol of w, and $ch(w)$ is the number of children of the last node of w. In other words, $P(\cdot \mid s_i \cap \mathcal{W}_c)$ is the steady probability of the input being in the state s_i and the compressed feed being in the chapter $\mathcal{W}_c$. Thus, the non-conditioned probability of w equals

$$P_c(w) = \sum_{i=1}^{N-1} P(s_i \cap \mathcal{W}_c) \cdot P(w \mid s_i \cap \mathcal{W}_c). \tag{1}$$

3.2 Dictionary generation

As previously stated, the dictionary $\mathcal{W}$ is built by concatenating 2^O consecutive chapters $\mathcal{W}_c$. We start with a naive initialization for the state/chapter probabilities, $\hat{P}_0(s_i \cap \mathcal{W}_c)$, that we will then refine iteratively. In all tested cases, the proposed method converges in just a few iterations and is extremely robust to different initialization conditions, thus we choose a simple initialization:

$$\hat{P}_0(s_i \cap \mathcal{W}_c) = \begin{cases} 1/2^O & \text{if } i = 1 \\ 0 & \text{if } i > 1 \end{cases}$$

Using this initialization we create 2^O chapters, arrange the words in some particular order, and then concatenate the chapters to form the dictionary.

Given a dictionary, we can calculate the steady probability of each state using a Markov model, as described in [1], but due to the fast convergence rate of the process this is unnecessary. We simply estimate $\hat{P}_l(w)$ for each w in $\mathcal{W}$ using Eqn. (1).

Then, we estimate the state/chapter probabilities iteratively by:

$$\hat{P}_{l+1}(s_i \cap \mathcal{W}_c) = \sum_{\substack{ch(w)=i \\ ov(w)=c}}^{w \in \mathcal{W}} \hat{P}_l(w)$$

where $ch(w)$ is again the number of children of the last node of w, and $ov(w)$ is the number represented by the O lower bits of the codeword corresponding to w.

3.3 Specialized chapters

A main source of inefficiency in the original Marlin is that one dictionary must be able of encoding the input, independently of its current state. As an example, let us suppose a single chapter dictionary where half of the words start with the symbol a_1. Now imagine that the source is currently in state s_2. As we know that the following word can not begin with a_1, the codeword emitted will be 1 bit larger than necessary.

By having overlapping codewords, we have several chapters, and we can specialize each chapter towards a specific state. Unlike previous multi-dictionary approaches [13, 14, 15, 16], we can not match perfectly one chapter to each state. This is because all chapters must have the same number of words, but the number of words that end in any particular state is variable.

The problem of optimally matching words to chapters is non-trivial, therefore we suggest a heuristic matching strategy. We build the chapters as described previously, and, to manipulate their indexes, we sort their words in ascending order by state, and within the words that belong to the same state, we sort them in descending order of probability. Then, we arrange the codewords such that the first words (lower state, higher probability) are assigned to the first chapter, the following words to the second chapter, and so on. As a result, it happens often that the first chapters will always receive the input in the s_1 state, while the last chapters would rarely need to represent the first symbols of the alphabet at the beginning of the word. An example of the distribution in specialized chapters is given in Fig. 2.

3.4 The victim chapter

The *victim chapter* is the second method we use to take advantage of having multiple dictionaries. The victim chapter is the only chapter that is designed to provide a non-empty match with the input no matter its contents. Thanks to the victim chapter, the other chapters do not waste words on symbols that have little chances of appearing.

If the current chapter can not produce a non-empty match for the input, it will emit the code for the empty word whose overlap bits reference the victim chapter. As a result, rare symbols will require at most two codewords to be encoded.

We select as a victim the least probable chapter (*i.e.*, the one that already contains the rarest words), this way we maximize the benefit of the other chapters. While building non-victim chapters, words whose probability is below a fixed threshold do not count towards the word limit of the chapter, and are discarded. The compression efficiency is not very sensitive to this threshold, and we use a value of $P_{th} = 0.01 \cdot 2^{-K}$.

$K = 3$. $O = 2$. $A = \{a, b, c, d, e, f, g\}$. $P(A) = \{0.4998, 0.2499, 0.2499, 10^{-4}, 10^{-4}, 10^{-4}, 10^{-4}\}$.

Initialization:

$\times 10^{-3}$	$\mathcal{W}_1$	$\mathcal{W}_2$	$\mathcal{W}_3$	$\mathcal{W}_4$
s_1	250	250	250	250
s_2	0	0	0	0
s_3	0	0	0	0
$\cdots$	0	0	0	0

(a) State Probabilities

$\varnothing \to s_1, \mathcal{W}_1$	$\varnothing \to s_1, \mathcal{W}_1$	$\varnothing \to s_1, \mathcal{W}_1$	$\varnothing \to s_1, \mathcal{W}_1$
$b \to s_1, \mathcal{W}_2$	$ab \to s_1, \mathcal{W}_2$	$ab \to s_1, \mathcal{W}_2$	$ab \to s_1, \mathcal{W}_2$
$d \to s_1, \mathcal{W}_3$	$ca \to s_1, \mathcal{W}_3$	$ca \to s_1, \mathcal{W}_3$	$ca \to s_1, \mathcal{W}_3$
$f \to s_1, \mathcal{W}_4$	$c \to s_2, \mathcal{W}_4$	$c \to s_2, \mathcal{W}_4$	$c \to s_2, \mathcal{W}_4$
$a \to s_1, \mathcal{W}_1$	$aa \to s_1, \mathcal{W}_1$	$aa \to s_1, \mathcal{W}_1$	$aa \to s_1, \mathcal{W}_1$
$c \to s_1, \mathcal{W}_2$	$ba \to s_1, \mathcal{W}_2$	$ba \to s_1, \mathcal{W}_2$	$ba \to s_1, \mathcal{W}_2$
$e \to s_1, \mathcal{W}_3$	$b \to s_2, \mathcal{W}_3$	$b \to s_2, \mathcal{W}_3$	$b \to s_2, \mathcal{W}_3$
$g \to s_1, \mathcal{W}_4$	$a \to s_3, \mathcal{W}_4$	$a \to s_3, \mathcal{W}_4$	$a \to s_3, \mathcal{W}_4$
(b) $\mathcal{W}_1$	(c) $\mathcal{W}_2$	(d) $\mathcal{W}_3$	(e) $\mathcal{W}_4$

First (and final) iteration:

$\times 10^{-3}$	$\mathcal{W}_1$	$\mathcal{W}_2$	$\mathcal{W}_3$	$\mathcal{W}_4$
s_1	312	312	94	0
s_2	0	0	94	94
s_3	0	0	0	94
$\cdots$	0	0	0	0

(f) State Probabilities

$aa \to s_1, \mathcal{W}_1$	$aa \to s_1, \mathcal{W}_1$	$b \to s_1, \mathcal{W}_1$	$cb \to s_1, \mathcal{W}_1$
$ba \to s_1, \mathcal{W}_2$	$ba \to s_1, \mathcal{W}_2$	$a \to s_1, \mathcal{W}_2$	$caa \to s_1, \mathcal{W}_2$
$\varnothing \to s_1, \mathcal{W}_3$	$\varnothing \to s_1, \mathcal{W}_3$	$\varnothing \to s_1, \mathcal{W}_3$	$\varnothing \to s_1, \mathcal{W}_3$
$c \to s_2, \mathcal{W}_4$	$c \to s_2, \mathcal{W}_4$	$f \to s_1, \mathcal{W}_4$	$b \to s_2, \mathcal{W}_4$
$ab \to s_1, \mathcal{W}_1$	$ab \to s_1, \mathcal{W}_1$	$c \to s_1, \mathcal{W}_1$	$cc \to s_1, \mathcal{W}_1$
$ca \to s_1, \mathcal{W}_2$	$ca \to s_1, \mathcal{W}_2$	$d \to s_1, \mathcal{W}_2$	$ba \to s_1, \mathcal{W}_2$
$b \to s_2, \mathcal{W}_3$	$b \to s_2, \mathcal{W}_3$	$e \to s_1, \mathcal{W}_3$	$cab \to s_1, \mathcal{W}_3$
$a \to s_3, \mathcal{W}_4$	$a \to s_3, \mathcal{W}_4$	$g \to s_1, \mathcal{W}_4$	$ca \to s_3, \mathcal{W}_4$
(g) $\mathcal{W}_1$	(h) $\mathcal{W}_2$	(i) $\mathcal{W}_3$	(j) $\mathcal{W}_4$

Figure 2: Naive state probabilities (a) are used to create the initial chapters (b-e). Then state probability estimates are updated (f), and new chapters are created (g-j). Words are represented pointing to the state they leave the input, and the next chapter. (b) and (i) are the victim *chapters. Words leaving the input in state 2 point to chapters 3 or 4. Words leaving the input in state 3 point to chapter 4. No words leaving the input in state 1 point to chapter 4, therefore chapter 4 has no words starting with 'a'. (g) and (h) are the same.*

3.5 Leveraging repeated chapters

The main drawback of using overlap is that the dictionary is larger than if no overlap were used, and larger means slower as a smaller portion of it will fit into the L1 cache. Hence the question whether it is better to use codewords with overlapping or a single large dictionary. The simple answer is that it depends: low entropy sources benefit significantly from overlapping, but high entropy sources prefer one large dictionary.

However, simply comparing dictionary sizes is not fair. In a single large dictionary, all words are different. On the other hand, in an overlapping dictionary organized in chapters, many of the words are repeated. In fact, it often occurs that entire chapters are just the same. We can leverage this fact to map different chapters to the same physical memory addresses. This mapping can be performed for free by utilizing the Translation Lookaside Buffer (TLB) integrated in all modern CPUs, as seen in Fig. 3. However, the size of the TLB (typically 128 entries) limits the number of chapters and thus the amount of overlap that can be used in practical terms.

In Fig. 4 we can see how overlapping has better efficiency per word for low entropy sources, while it is as efficient as a large dictionary for high entropy sources.

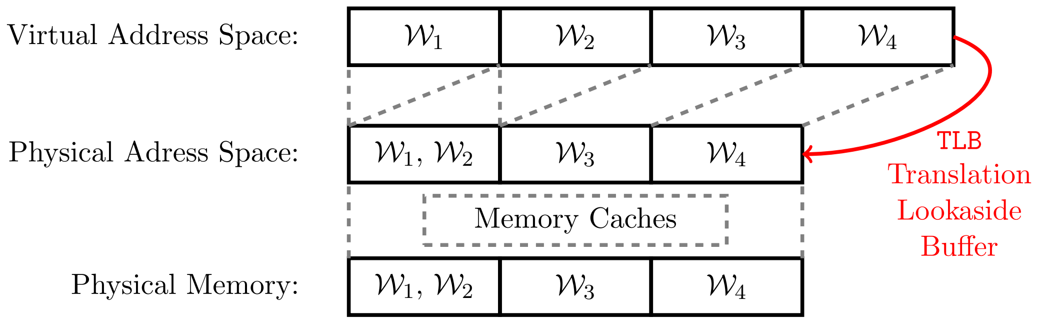
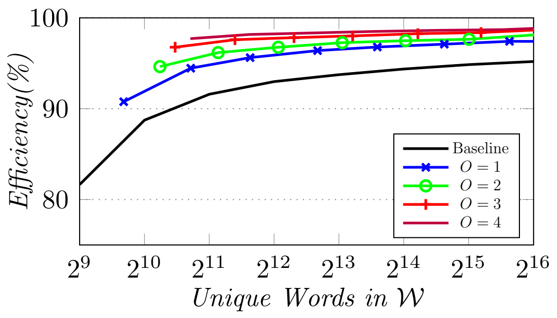
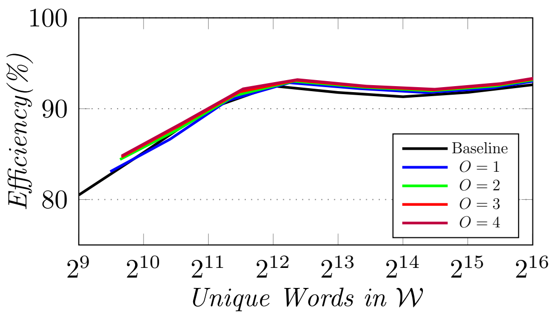


Figure 3: Chapters $\mathcal{W}_1$ and $\mathcal{W}_2$ in Fig. 2 are the same and they can be stored in the same region of physical memory. Marlin sees the entire dictionary as a single table with all chapters concatenated (in the virtual address space*). The Translation Lookaside Buffer (TLB) is a hardware component in CPUs that translates* virtual addresses *into* physical addresses *where the data is stored in memory, and it can be used to provide deduplication at no additional cost. As the TLB acts before the memory caches, the deduplication increases the effectiveness of the L1 cache.*



(a) Efficiency vs. Size. Laplacian. $H = 25\%$.



(b) Efficiency vs. Size. Laplacian. $H = 75\%$.

Figure 4: Low entropy sources benefit more from overlapping than high entropy sources. However, compared to having a single large dictionary, overlapping never wastes space.

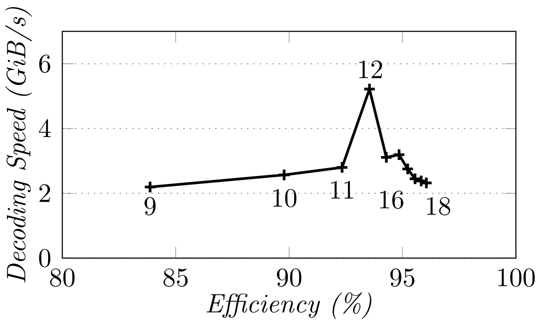
4 Evaluation

We implemented Marlin using the presented overlapping codewords approach, and the code is made publicly available[1]. 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. To ensure reproducible performance measurements, we perform each test once without measuring it, and then we test it again as many times as possible during (at least) one second and we report the average time per test. Standard deviation is not significant ($\sigma < 1\%$). Unless stated otherwise, the test data blobs are 2^{20} symbols from a 8 bit Laplacian source with an entropy of 50%.

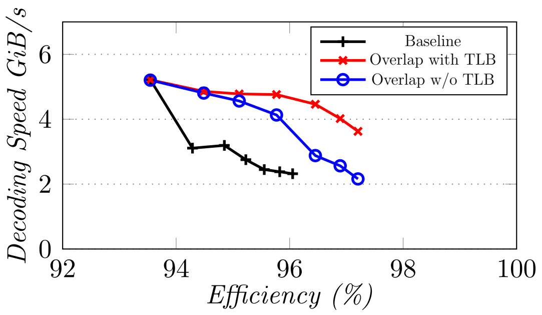
Our main evaluation metric is the *compression efficiency* defined as the ratio between the information entropy of the source and the average bit rate achieved: $\eta_X = H(X)/\text{ABR}(X)$. To evaluate coding throughput we use the unit GiB/s.

[1]Git repository: `https://github.com/MartinezTorres/marlin` (branch:`dcc2018`)

We suggest to use $K = 12$. This value hits a double sweet spot, as $|\mathcal{W}| = 2^{12}$ is small enough for the entire dictionary to fit into the L1 cache, and the least common multiple between 12 bits and 1 byte is just 3 bytes, allowing us to unroll the entire decoding loop in a branch-less fashion. Therefore, $K = 12$ is almost twice as fast when compared to other values of K, as seen in Fig. 5a. Regarding overlap, we recommend $O = 4$, as larger values affect the decoding throughput, as seen in Fig. 5b.



(a) Speed vs. Efficiency. Baseline.

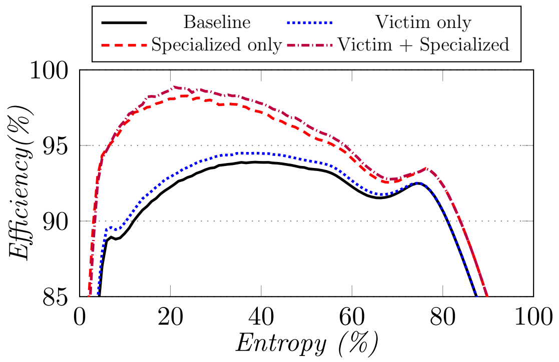


(b) Speed vs. Efficiency. Impact of TLB.

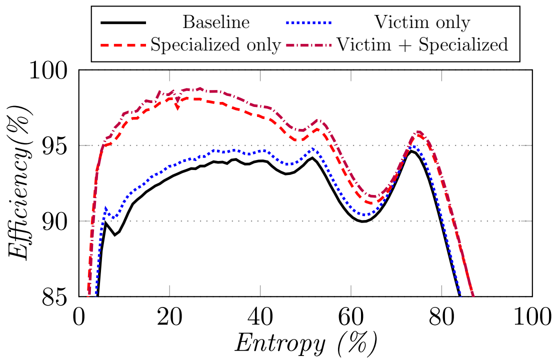
Figure 5: Laplacian source of $H = 50\%$. Left: Speed vs. Efficiency curve of non-overlapping Marlin for different values of K. $K = 12$ is the sweet spot. Right: we show the performance of Marlin with $K = 12$ and different values of overlapping ($0 \leq O \leq 6$). We see how TLB based deduplication helps to obtain a faster operation.

4.1 Synthetic data results

In Fig. 6 we evaluate the two mechanisms we use to provide compression gains from overlapping: the chapter specialization and the victim chapter.



(a) Efficiency on Laplacian distributions



(b) Efficiency on Gaussian distributions

Figure 6: $K = 12$ and $O = 4$. Efficiency of overlapping on Laplacian and Gaussian distributions. The largest benefit comes from using specialized chapters.

Chapter specialization offers the largest performance improvement and allows to raise the efficiency of Marlin above 95% for the lower entropy sources. The victim chapter allows us not to waste words in rare symbols in all but one chapter, therefore, its benefit is proportional to the ratio between the number of rare symbols in the alphabet and the chapter size. For $K = 12$ its effect is not pronounced, but nevertheless it increases compression efficiency by approximately an extra 0.5%.

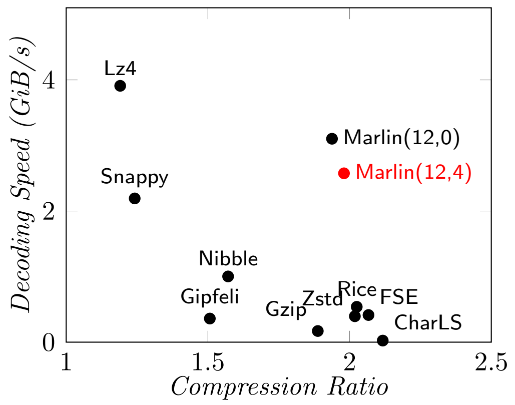
4.2 Real data results

Lossless image compression is a natural application for entropy codecs. We evaluate the impact of overlapping codewords on the Rawzor [18] 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. The Marlin compressor uses 11 predefined dictionaries, as explained in detail in [1].

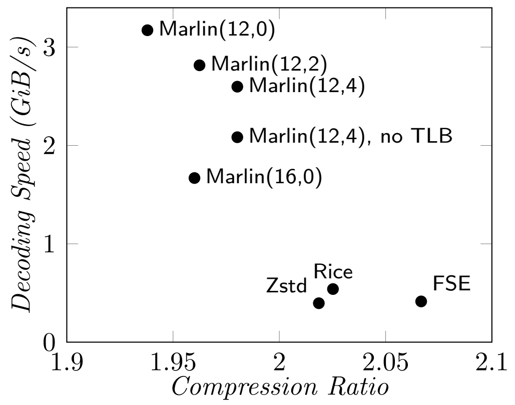
We can see how Marlin compares to several state-of-the-art codecs in Fig. 7a. In this dataset, naive Marlin [1] achieves a compression ratio of 1.937. Using an overlap of 4 bits shifts the compression ratio to 1.980, which corresponds to an improvement of 2.22%. As expected, the encoding speed drops from 114.6 MiB/s to 82.6 MiB/s and is still faster than gzip [19] that encodes at 64.2 MiB/s. More importantly, the decoding speed drops from 3171.52 MiB/s to 2597 MiB/s, and still Marlin remains faster than Snappy [5], a reference HT algorithm that achieves 2220 MiB/s here.

This drop in decoding speed is larger than the one observed on our experiments using synthetic data in Fig. 5b. This is because here we used a smaller block size (4 KiB vs. 1 MiB), thus the TLB initialization process that happens once per block has a larger impact. We will investigate methods to alleviate this penalty.

In Fig. 7b we can see how using the TLB boosts the decoding speed by 24.7%. We can also see how simply increasing the dictionary size to 2^{16} instead of using overlapping has a larger penalty in speed, while not achieving corresponding gains in compression efficiency.



(a) RAW Image Compression



(b) RAW Image Compression (detail)

Figure 7: Evaluation on the Rawzor image compression dataset. Notation: Marlin(K,O). Left: Overlap affects the throughput of Marlin, but it is still faster than Snappy, a reputable HT algorithm. Right: Detailed view. Overlap provides better compression ratio and less decoding speed penalty than using larger dictionaries. Note how the utilization of the TLB creates a significant benefit for the throughput.

5 Conclusions

We have presented a method named Partially Overlapping Codewords that allows us to encode information regarding the state of the input source in Variable-to-Fixed codes within the codeword bits. Combined with plurally parsable dictionaries, whose compression efficiency depends on knowing the current state of the input, it allows us to create an entropy codec that is both efficient and fast. We have applied this technique on the Marlin codec, boosting its efficiency in low entropy distributions up to 98.6%, thus improving its performance as a High Throughput entropy codec.

References

[1] M. Martinez, M. Haurilet, R. Stiefelhagen, and J. Serra-Sagristà, "Marlin: A high throughput variable-to-fixed codec using plurally parsable dictionaries," in *Proceedings of Data Compression Conference*. IEEE, 2017, pp. 161–170.

[2] 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.

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

[4] 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.

[5] Z. Tarantov and S. Gunderson, "Snappy," google.github.io/snappy, 2011, [Accessed 28-October-2017].

[6] Y. Collet, "LZ4," lz4.github.io/lz4, 2011, [Accessed 28-October-2017].

[7] M. Oberhumer, "LZO: Lempel Zip Oberhumer," www.oberhumer.com/opensource/lzo, 1996, [Accessed 28-October-2017].

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

[9] 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.

[10] J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM Journal of Research and Development*, vol. 20, no. 3, pp. 198–203, 1976.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

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

[18] S. Garg, "The new test images," www.imagecompression.info/test_images, 2011, [Accessed 28-October-2017].

[19] J.-l. Gailly, "Gzip," www.gnu.org/software/gzip, 1992, [Accessed 28-October-2017].