IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received March 5, 2020, accepted April 27, 2020, date of publication April 29, 2020, date of current version May 14, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2991314

High-Throughput Variable-to-Fixed Entropy Codec Using Selective, Stochastic Code Forests

MANUEL MARTÍNEZ TORRES[1], MIGUEL HERNÁNDEZ-CABRONERO[2], IAN BLANES[2], (Senior Member, IEEE), AND JOAN SERRA-SAGRISTÀ[2], (Senior Member, IEEE)

[1]Karlsruhe Institute of Technology, 76131 Karlsruhe, Germany
[2]Information and Communications Engineering, Universitat Autònoma de Barcelona, 08193 Bellaterra, Spain

Corresponding author: Miguel Hernández-Cabronero (miguel.hernandez@uab.cat)

This was supported in part by the Postdoctoral Fellowship Program Beatriu de Pinós through the Secretary of Universities and Research (Government of Catalonia) under Grant 2018-BP-00008, in part by the Horizon 2020 Program of Research and Innovation of the European Union under the Marie Skłodowska-Curie under Grant 801370, in part by the Spanish Government under Grant RTI2018-095287-B-I00, and in part by the Catalan Government under Grant 2017SGR-463.

ABSTRACT Efficient high-throughput (HT) compression algorithms are paramount to meet the stringent constraints of present and upcoming data storage, processing, and transmission systems. In particular, latency, bandwidth and energy requirements are critical for those systems. Most HT codecs are designed to maximize compression speed, and secondarily to minimize compressed lengths. On the other hand, decompression speed is often equally or more critical than compression speed, especially in scenarios where decompression is performed multiple times and/or at critical parts of a system. In this work, an algorithm to design variable-to-fixed (VF) codes is proposed that prioritizes decompression speed. Stationary Markov analysis is employed to generate multiple, jointly optimized codes (denoted code forests). Their average compression efficiency is on par with the state of the art in VF codes, e.g., within 1% of Yamamoto *et al.*'s algorithm. The proposed code forest structure enables the implementation of highly efficient codecs, with decompression speeds 3.8 times faster than other state-of-the-art HT entropy codecs with equal or better compression ratios for natural data sources. Compared to these HT codecs, the proposed forests yields similar compression efficiency and speeds.

INDEX TERMS Data compression, high-throughput entropy coding, variable-to-fixed codes.

I. INTRODUCTION

High throughput (HT) data compression is widely employed to improve performance in many systems with strong time constraints. In communications applications, compression improves effective channel capacity [1], [2], and HT is essential to avoid undesired transmission delays. Large-scale data processing applications benefit from HT compression, e.g., to reduce network, disk and memory usage in computer clusters and distributed (including cloud) storage systems [3]–[10].

HT codecs produce compact representations of input data, prioritizing the rate at which samples are processed over the achieved *compression ratios* –i.e., the quotient between the original and compressed data sizes. Even though HT can be attained via massive parallelization [11]–[14], low-complexity compression pipelines are often employed for this purpose [15]–[25]. Lower complexity typically entails lower power consumption, and therefore HT codecs are also suitable in scenarios with energy consumption and delay constraints. These include remote sensing and earth observation onboard satellites [26], [27], real-time transmission of point clouds for search and rescue [28], cooperative robot coordination [29], and Internet of Things (IoT) scenarios [30]–[32]. Many other applications can benefit from time-efficient and energy-efficient data compression, in particular when involving interactive communication and mobile devices. The use of compression in hypertext transfer protocol data [33], [34] and in wireless sensor networks [35] are two notable examples thereof.

In many of the applications described above, *decompression* performance (in terms of time and energy consumption) is equally as or more important than compression performance. Decompression time is critical to minimize delay in high-performance network contexts [1], [2], [36],

The associate editor coordinating the review of this manuscript and approving it for publication was Jenny Mahoney.

in particular when headers need to be obtained for routing and other purposes in intermediate nodes [1], [32], [35], and to allow fast access to data in cloud and distributed file systems [5], [37]–[39]. In the context of high-performance computing, it is paramount to quickly retrieve compressed data [3], [4], [9], especially in data-driven applications [6], [36], [40] using distributed databases and compressed columns. Furthermore, minimizing energy spent at the decoder can extend the availability of battery-powered devices, e.g., for mobile networks and IoT [32], [32], [35], robot coordination in rescue situations [28] and mobile web browsing [34], to name a few. Power efficiency in the decoding process also has a positive effect on the carbon footprint of data centers where it is intensively applied [1]–[5], [9], [37]–[39], notably when the same data are typically decoded at different points in time [6], [40].

Variable-to-fixed (VF) coding produces fixed-length words, each representing multiple input symbols [41]–[53]. In this work, we present an HT compressor-decompressor pair based on VF codes, designed to provide significantly higher decoding speed while yielding compression ratios and coding speeds comparable to state-of-the-art HT entropy codecs. A method is first proposed to create dictionaries (equivalent to code trees) stochastically optimized for any given symbol probability distribution. An extension using code forests allows adding a parametrizable number of coding contexts, enabling configurable time-compression ratio trade-offs. Further improvements are proposed based on selectively dedicating more effort to more compressible parts of the input data. This enables the design of more efficient codes and minimizes the impact of low-probability symbols and of incompressible noise. The proposed codec extends upon our previous conference works [21], [54] with theoretical contributions, exhaustive experimental results to assess their performance, and a standalone code container released with digital object identifier (DOI) 10.24433/CO.2752092.v2[1] as supplementary materials to enable full experimental reproducibility.

The remainder of this paper is structured as follows: Section II discusses the closest related works in the state of the art. Section III provides definitions and a concise review of the VF coding literature. The proposed HT codec is described in Section IV, and exhaustive experimental evaluation is presented in Section V. Section VI concludes this work.

II. RELATED WORK

Some of the best-performing HT algorithms in the state of the art are based on a Lempel-Ziv (LZ) [55] decomposition, and/or a fast entropy coding stage. LZ decomposition removes data redundancy by expressing the input as a series of position-length references to a dynamically built dictionary [55]. These references can be directly output –e.g., as in LZ4 [15] and Google's Snappy [16]–, compressed with fast bit mangling techniques [17], encoded with a fast version of Huffman's entropy coder –e.g., as in LZO [18] and Gipfeli [19]–, or coded with a combination of fast Huffman and asymmetrical numeral system (ANS) [56] algorithms, for example as in Zstandard [20]. Random data sources –e.g., with Laplacian distributions– appear naturally when dealing with image, audio and other sensor signals when using differential pulse-code modulation (DPCM) and other common prediction and/or transform-based methods. For these and other random sources, LZ decomposition does not yield good results, and especially low efficiency has been reported for medium and low source entropy rates [21]. Competitive codecs have been proposed that obviate any LZ decomposition and rely on a fast entropy codec. A modified version of Huffman is used in Huff0 [22]. Finite State Entropy (FSE) [23] provides a fast implementation of ANS; and Rice codes are used in fast JPEG-LS implementations such as CharLS [24]. FAPEC [25], a recent compression algorithm designed for space missions, includes an HT entropy codec that is adaptively applied to blocks of prediction errors.

In all cases above,[2] HT compression is attained by repeatedly coding one [22]–[25] or several [18]–[20], [56] input symbols into variable-length words. Even though these compression schemes enable fast encoding and reasonable compression ratios, one-pass decoders cannot know a priori the boundaries between coded words [15]–[25]. Therefore, they must continually execute conditional branches that hinder efficient pipelining of the decoding process, thus limiting maximum decoding speed and increasing energy consumption.

By construction, VF codes can be decoded very efficiently without requiring conditional branches at the most critical parts of the decoding loop. Tunstall proposed an algorithm that produces codes optimal within a subclass of all possible VF codes called *prefix free* [41]. Since then, more general classes of VF codes have been described that improve upon Tunstall. Savari introduced the concept of plurally parsable dictionaries, and proposed a generation algorithm for the binary case [44]. Yamamoto and Yokoo described the class of plurally parsable codes called almost-instantaneous variable-to-fixed (AIVF) codes, which can also be employed in the non-binary case [47]. Some improvements have been proposed to [47], reporting small gains in terms of compressed sizes [49], [51] and memory efficiency [52]. This work contributes with a new method for constructing VF codes that provide an advantageous trade-off between compression efficiency and computational complexity, specifically for the decoder. It should be highlighted that other variable-to-fixed codecs not discussed in this work have been proposed in the literature, e.g., [42], [43], [45], [46], [48], [52]. However, these are based on arithmetic coding as opposed to dictionary-based coding, which enables higher (better) compression ratios but at a prohibitive complexity cost in the scope of HT codecs.

[1] Also available at https://codeocean.com/capsule/9466901/.

[2] Except arguably for [15] and [16], which can be considered variable-to-fixed codecs since they output fixed-length LZ dictionary references.

III. BACKGROUND

A. DEFINITIONS AND NOTATION

1) SOURCES, SEQUENCES AND CODECS

Let $\aleph$ be a discrete source that produces symbols from a finite alphabet $\mathcal{A} = \{a_1, \ldots, a_{|\mathcal{A}|}\}$. When the probability of emission of the symbols is known, $\mathcal{A}$ also denotes a sequence sorted in decreasing order of probability. Let $\mathcal{A}^+$ be the set of all finite, nonempty sequences of elements in $\mathcal{A}$. Elements $\mathbf{W} \in \mathcal{A}^+$ are referred to as *words* and denoted as $\mathbf{W} = w_1\, w_2 \cdots w_{|\mathbf{W}|}$, where $w_i \in \mathcal{A}$ denotes the i-th symbol in the word. Let $\mathcal{B}^+$ be the set of all finite, nonempty binary sequences. A lossless entropy codec can be defined as a pair of coder and decoder functions $\left(\mathcal{E} : \mathcal{A}^+ \longrightarrow \mathcal{B}^+,\ \mathcal{D} : \mathcal{B}^+ \longrightarrow \mathcal{A}^+\right)$ that satisfy $\mathcal{D}(\mathcal{E}(\mathbf{A})) = \mathbf{A}$ for all $\mathbf{A} \in \mathcal{A}^+$. Hereafter, lossless entropy codecs as per this definition are considered exclusively.

2) VARIABLE-TO-FIXED DICTIONARIES

Variable-to-fixed coders split $\mathbf{A} \in \mathcal{A}^+$ into a finite sequence of words whose concatenation yields $\mathbf{A}$. Each word is then represented by a binary sequence of fixed length $l > 0$, and $\mathcal{E}(\mathbf{A}) \in \mathcal{B}^+$ is defined as the concatenation of all binary sequences. Each l-bit sequence can be considered an index to a *dictionary* $\mathcal{W}$ that contains at most 2^l words and unambiguously describes its associated variable-to-fixed code.

VF decoders can be implemented very efficiently by iteratively reading l bits from $\mathcal{E}(\mathbf{A})$ and outputting the corresponding word's symbols. Dictionaries associated with VF codes can be defined a priori, or dynamically constructed as input samples are processed, e.g., with the LZ algorithm [55]. A dictionary $\mathcal{W}$ fully defines a codec assuming the criterion of using the longest word possible. This criterion is general in the literature and is embraced as well in this work. In what follows, only *complete* dictionaries are considered, i.e., for any valid input sequence $\mathbf{A} \in \mathcal{A}^+$, there exists at least one concatenation of words in $\mathcal{W}$ such that $\mathbf{A}$ is a prefix of that concatenation.

The statistical properties of $\aleph$, in particular the probability of emission of each $a \in \mathcal{A}$, can be exploited to optimize dictionary performance. Sources are hereafter assumed to be ergodic –e.g., stationary– with symbol probability distribution P. As discussed in Section I, probability-driven design enables more efficient compression than LZ-based methods for sources well modeled by random variables, e.g., following a Laplacian distribution [21]. Therefore, this approach to dictionary design is considered exclusively hereinafter.

3) CODE TREES

VF dictionaries can always be expressed as a *code tree*, in which each edge is assigned a symbol $a \in \mathcal{A}$. Each node is associated with a word $\mathbf{W}$ formed by the symbols associated with the edges of its path from the root node, and a *present* flag determining whether that word is *Included* or *Excluded* from $\mathcal{W}$. The code tree $\mathcal{T}$ equivalent to a dictionary $\mathcal{W}$ is defined as the one that satisfies these conditions:

Algorithm 1 VF Coding of an Input Sequence $\mathbf{A}$ Using a Code Tree $\mathcal{T}$

```
1:  function TreeCoding(T, A)
2:      n ← T.root                          ▷ n: current node
3:      for each a ∈ A                      ▷ a: current input symbol
4:          if n.HasEdge(a) then
                // Follow edge associated with a
5:              n ← n.Child(a)
6:          else
                // Word defined by the path from T.root
7:              EmitWord(n.W)
8:              n ← T.root.Child(a)
9:          end if
10:     end for
11:     if n ≠ T.root then
12:         FlushLastSymbols(n)
13:     end if
14: end function
```

1) For each word $\mathbf{W}$ in the dictionary, there exists exactly one node with associated word $\mathbf{W}$ and *present* flag set to *Included*.
2) All *Included* nodes have words included in $\mathcal{W}$.
3) The *present* flag is *Included* for all leaf nodes. A node is a leaf when it has no children.
4) No two nodes can have the same associated word.

The encoding process using a code tree is described in Algorithm 1. It starts at the root node, and each input symbol is processed sequentially. If the current node has an edge with associated symbol equal to the input, and the edge connects a child node, the algorithm advances to that child node. Else, the algorithm emits the current node's word $\mathbf{W}$, i.e., it appends that word's index binary representation to the current output. Any one-to-one mapping between the words and the available binary indices can be used, as long as it is known by both the coder and the decoder. A routine called EmitWord is assumed to be available in this and subsequent algorithms to invoke this functionality. After emission of a word, the algorithm returns to $\mathcal{W}$'s root and the process is repeated. If the dictionary completeness assumption described above holds, nodes associated to emitted words are guaranteed to be *Included* in $\mathcal{W}$. After the main coding loop, one or more symbols might remain uncoded. By construction, these symbols are prefix of some *Included* words in $\mathcal{W}$. The FlushLastSymbols function emits any of those words and sends as side information the number of symbols coded this way.

A common decompression algorithm exists for all VF codes. Each emitted fixed-length index identifies a single *Included* word, that can be looked up in a table that contains that word's symbols. In case the FlushLastSymbols function emitted a word during compression, the decoder truncates that word to match the decompressed symbol sequence with the original one. Side information needed to perform this truncation is negligible for long enough sequences and not considered hereafter.

Algorithm 2 Tunstall's Algorithm for Creating an Optimal Prefix-Free Dictionary for Alphabet $\mathcal{A}$ and Symbol Probability P of up to max_size Words

1: **function** Tunstall($\mathcal{A}$, P, max_size)
   // Initialization
2: $\mathcal{W} \leftarrow \{\mathbf{W}_a\}_{a\in\mathcal{A}}, \mathbf{W}_a = a$
   // Node expansion
3: **while** $|\mathcal{W}| + |\mathcal{A}| \leq$ max_size **do**
4: $\mathbf{W} \leftarrow \arg\max_{\mathbf{W}\in\mathcal{W}} \{P_{\text{word}}(\mathbf{W})\}$
5: $\mathcal{W} \leftarrow \mathcal{W} \setminus \{\mathbf{W}\}$ ▷ \: set subtraction
6: **for each** $a \in \mathcal{A}$
7: $\mathbf{W}' \leftarrow \mathbf{W} \oplus a$ ▷ $\oplus$: concatenation
8: $\mathcal{W} \leftarrow \mathcal{W} \bigcup \{\mathbf{W}'\}$ ▷ $\bigcup$: set union
9: **end for**
10: **end while**
11: **return** $\mathcal{W}$
12: **end function**

B. PREFIX-FREE DICTIONARIES

A dictionary $\mathcal{W}$ is prefix-free (or proper) if it satisfies $\forall \mathbf{W}, \mathbf{W}' \in \mathcal{W}, \mathbf{W} \neq w'_1 w'_2 \cdots w'_{|\mathbf{W}|}$, i.e., no word is the beginning of another. Under this constraint, no valid input $\mathbf{A} \in \mathcal{A}^+$ can be expressed by more than one concatenation of words in $\mathcal{W}$. Tunstall [41] proposed a method to design optimal prefix-free dictionaries assuming a memoryless source $\aleph$, i.e., one that produces symbols independently. Tunstall's method is listed in Algorithm 2. First, the dictionary is initialized with $|\mathcal{A}|$ words, each one representing a different input symbol (line 2). In successive iterations, each word $\mathbf{W} \in \mathcal{W}$ is assigned a probability

$$P_{\text{word}}(\mathbf{W}) = \prod_{w\in\mathbf{W}} P(w), \tag{1}$$

where w iterates over $\mathbf{W}$'s symbols. The most probable word $\mathbf{W}$ based on P_{word} is then substituted by $|\mathcal{A}|$ new words, each resulting from concatenating $\mathbf{W}$ and a different symbol from $\mathcal{A}$ (lines 4 to 10). The algorithm stops when no more words can be expanded without exceeding the specified maximum number of words.

Using code tree notation, each iteration of Tunstall's algorithm can be regarded as adding $|\mathcal{A}|$ children to the leaf node with the highest estimated probability. To satisfy the prefix-free constraint, only words represented by leaf nodes are included in the output dictionary $\mathcal{W}$. Fig. 1 shows an example of an iteration of Tunstall's algorithm using code tree notation for $\mathcal{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}\}$ and associated probabilities 0.8, 0.1, 0.07, and 0.03. In this example, the words included in $\mathcal{W}$ are {aa, ab, ac, ad, b, c, d}, and the node to be expanded in the next iteration (not shown in the figure) is that with $\mathbf{W} = \mathsf{aa}$ and associated probability 0.64.

C. PLURALLY PARSABLE DICTIONARIES

The optimality of Tunstall codes is restricted to the prefix-free case. More general, plurally parsable (or non-proper) dictionaries that remove the prefix-free constraint were introduced in [44]. Using tree notation, words associated with non-leaf nodes can also be included in $\mathcal{W}$ when removing the prefix-free constraint. This has enabled more flexible and efficient designs [47], [49]–[51], [53].

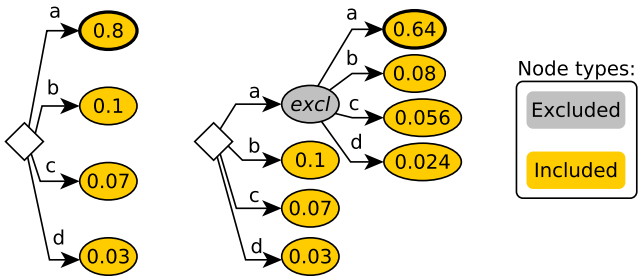

FIGURE 1. Tree representation of an iteration of Tunstall's algorithm: (left) tree initialization; (right) expansion of the most probable node. Nodes with maximum probability are highlighted with a thicker line in both depicted states.

In [44], Savari designs dictionaries that outperform Tunstall's for the case of predictable, memoryless binary sources, i.e., $\aleph$ producing symbols from $\mathcal{A} = \{0, 1\}$ independently with large $P(0)$ or large $P(1)$. She considers the encoding process as a Markov chain and analyzes its steady state to appraise dictionary performance, although her analysis applies only to the binary case and is described as cumbersome [44] and difficult to apply to the non-binary case [47]. Rababa *et al.* improve upon Savari's method by removing the constraint of always emitting the longest possible word [51]. Input can then be expressed using different word sequences with the same length as that obtained with the aforementioned constraint, and the choice made in the coding process is used as a side channel between the coder and the decoder.

In [47], a class of plurally parsable dictionaries called almost instantaneous variable to fixed (AIVF) codes is proposed. A complete dictionary $\mathcal{W}$ is AIVF if and only if its associated code tree satisfies the following conditions:

1) All leaf nodes in the code tree represent words included in $\mathcal{W}$.
2) Non-leaf nodes that have $|\mathcal{A}|$ children (complete nodes) represent words *not* included in $\mathcal{W}$.
3) Words represented by non-leaf nodes with fewer than $|\mathcal{A}|$ are included in $\mathcal{W}$.

The algorithm in [47] differs from Tunstall's in the fact that nodes can be partially expanded, leading to more flexible designs. This is illustrated with an example in Fig. 2 for $\mathcal{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}$ and associated probabilities {1/3, 1/4, 1/6, 3/20, 1/10}. In [47], multiple plurally parsable dictionaries are also proposed to exploit some dependencies between consecutively emitted words. Further analysis and improvements to [47] based on dynamic programming are proposed in [49], [53]. The code tree by [53] for the same example is shown in Fig. 2b.

IV. PROPOSED HT CODEC

A novel HT codec based on plurally parsable dictionaries is proposed in this section. An algorithm for creating individually optimized code trees is proposed in Section IV-A,

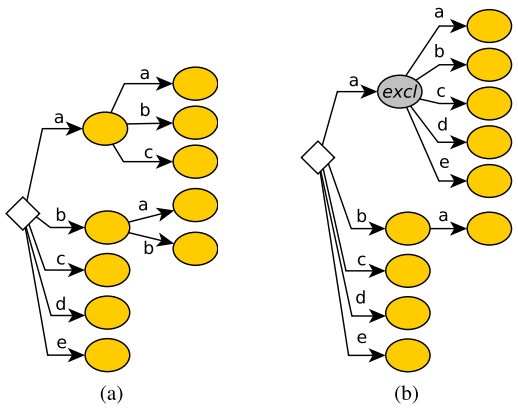

FIGURE 2. Example of plurally parsable code trees from [53]: *(a)* created using [47]; *(b)* created using [53].

based on stochastic optimization. In Section IV-B, multiple code trees are optimized jointly to generate code forests that improve upon individual code trees. Further improvements to the code forest generation algorithm are proposed in Section IV-C to prioritize resource allocation to the most compressible parts of the input.

A. STOCHASTICALLY OPTIMIZED CODE TREES

The description of the proposed algorithm for generating stochastically optimized code trees is presented in three parts. Section IV-A.1 defines the generation of basic trees. Section IV-A.2 models code trees as stochastic processes, enabling the creation of better adapted trees. Section IV-A.3 describes the iterative process used in the proposed codec to generate stochastically optimized code trees.

1) BASIC TREE GENERATION

Given a node $\mathbf{n}$ with associated word $\mathbf{W}$, the probability of the encoder emitting $\mathbf{W}$ is lower if that node has a nonempty set of children, hereafter denoted $children(\mathbf{n})$. This is because some occurrences of $\mathbf{W}$ in the sequence of input symbols will be coded by words associated with nodes contained in $children(\mathbf{n})$. Since longer words are preferred to maximize compression efficiency, a child word will be used whenever the next input symbol a can be matched, i.e., when an edge associated with a is connected to $\mathbf{n}$ in the code tree. Based on this, for the first coded word, the probability of a node being emitted can be calculated as

$$P_{\text{node}}(\mathbf{n}) = P_{\text{word}}(\mathbf{W}) \cdot \left(1 - \sum_{\mathbf{m} \in children(\mathbf{n})} P\left(symbol(\mathbf{m})\right)\right), \quad (2)$$

where $symbol(\mathbf{m})$ denotes the symbol of the edge connecting $\mathbf{m}$ and its parent $\mathbf{n}$. By definition, when $children(\mathbf{n})$ is empty, $P_{\text{node}}(\mathbf{n}) = P_{\text{word}}(\mathbf{W})$. As discussed in IV-A.2, P_{node} is not always accurate for the second and subsequent coded words. For the sake of simplicity, P_{node} is assumed to be exact for basic tree generation.

Algorithm 3 Basic Code Tree Generation Without Stochastic Optimization for Alphabet $\mathcal{A}$, Symbol Probability P and max_size *Included* Nodes

```
1:  function GetBasicTree(A, P, max_size)
2:      W ← {W_a}_{a∈A}, W_a = a
3:      T ← T_W                          ▷ Tree with the root and |A| children
4:      while |T.IncludedNodes()| ≤ max_size do
5:          n ← arg max_{n∈T.IncludedNodes()} {P_node(n)}
            // A's symbols sorted by decreasing probability.
6:          n.AddChild(a_{|children(n)|+1})
7:          if |children(n)| = |A| − 1 then
                // Last possible symbol
8:              n.AddChild(a_{|A|})
9:              n.present ← Excluded
10:         end if
11:     end while
12:     return T
13: end function
```

The proposed method is given in Algorithm 3. It follows a generalization of Tunstall's iterative algorithm, with the following three main differences:

1) In each iteration, the selected node is the one with the highest $P_{\text{node}}(\mathbf{n})$ instead of the highest $P_{\text{word}}(\mathbf{W})$ (line 5). This is hereafter referred to as the node being *expanded*.
2) Only one node $\mathbf{m}$ is created and connected to the expanded node $\mathbf{n}$ (line 6). The expanded node can be selected again in subsequent iterations. This is as opposed to Tunstall's algorithm, which connects all possible children to the node being expanded in a single iteration. Note that nodes added with AddChild are flagged as *Included* by default.
3) If an expanded node $\mathbf{n}$ is connected to $|\mathcal{A}|-1$ children, a second node $\mathbf{m}$ is connected to $\mathbf{n}$ associated to the last symbol in $\mathcal{A}$, $\mathsf{a}_{|\mathcal{A}|}$ (line 8). Then $\mathbf{n}$ is flagged as *Excluded*, removing its associated word from the dictionary (line 9). These changes can be safely applied after 2 by construction, because the emission of $\mathbf{n}$'s word $\mathbf{W}$ can only take place if symbol $\mathsf{a}_{|\mathcal{A}|}$ is produced after $\mathbf{W}$ by the source, or the end of the input sequence is reached after $\mathbf{W}$. Otherwise, by construction, another word would have been emitted. The changes benefit compression performance by increasing the average number of symbols per word included in the dictionary.

Algorithm 3 can be regarded as fast approximation to the average-sense optimal almost instantaneous VF code tree generation algorithm presented in [47]. In [47], sometimes all children of a node $\mathbf{n}$ are added instead of creating a single child in the node with highest associated probability. This occurs when the average word length gain due to the automatic expansion of $\mathbf{n}$'s last symbol outweighs creating otherwise suboptimal children. Empirically, we observed that this has little impact in most cases.

2) CODE TREES AND STOCHASTIC PROCESSES

Plurally parsable dictionaries introduce temporal dependencies between words. For instance, consider the code tree depicted in Fig. 2 for $\mathcal{A} = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}$. If word $\mathbf{W} = \mathsf{a}$ is emitted, since longer words are always preferred for coding, the next emitted word must start with symbols d or e. Otherwise, words aa, ab or ac would have been emitted instead of $\mathbf{W}$. Therefore, a tree obtained by direct application of Algorithm 3 will generally not be optimally adapted for the next symbol. This is because $P(\mathsf{a})$, $P(\mathsf{b})$ and $P(\mathsf{c})$ were not assumed to be 0 in its creation. In general, suboptimal probability estimation will occur after emitting any non-leaf's word. Algorithm 4 is proposed below based on stochastic analysis to address this issue and generate an improved tree, which maximizes expected *emitted* word length for a long-enough sequence of symbols. This is as opposed to considering only the expected word length of a dictionary, which is accurate to measure the dictionary's efficiency only in the prefix-free case due to the lack of temporal dependencies between words discussed above.

The coding procedure using a tree $\mathcal{T}$ is modeled as a stochastic process $X_{\mathcal{T}}$ with $|\mathcal{A}| - 1$ states, $\Sigma_{X_{\mathcal{T}}} = \{\sigma_1, \ldots, \sigma_{|\mathcal{A}|-1}\}$, which are transitioned whenever a word is emitted. When $\mathcal{T}$ is obvious from the context, X is used instead of $X_{\mathcal{T}}$. Hereafter, X is defined to be in state σ_i when $\mathcal{T}$'s structure and the last emitted word determine that the next coded word cannot start with any of the $i-1$ most probable symbols, i.e., the first $i-1$ symbols in $\mathcal{A}$. After emitting a leaf node's word, the process will transition to state σ_1 because there are no restrictions on the next input symbol. By design, Algorithm 3 adds children associated with symbols in decreasing order of probability, i.e., in the order given by $\mathcal{A}$. Therefore, after emitting the word associated with a node $\mathbf{n}$, X will be in state $\sigma_{|children(\mathbf{n})|+1}$. Using this definition, the probability of a node's word being emitted can now be expressed as

$$P_X(\mathbf{n}) = \sum_{\sigma \in \Sigma_X} P_X(\mathbf{n} \,|\, \sigma)\, \pi(\sigma), \tag{3}$$

where $\pi(\sigma)$ is the probability of X being in state σ before emitting a word, and $P_X(\mathbf{n} \,|\, \sigma)$ is that node's conditional probability, given that the process is in state σ before emission.[3] The new measure P_X can directly replace P_{node} in line 5 of Algorithm 3. The resulting function for producing trees given a finite discrete process X is hereafter denoted GetProcessTree($\mathcal{A}$, P, Σ_X, π, max_size), where Σ_X is X's state set and π is a function satisfying $\sum_{\sigma \in \Sigma_X} \pi(\sigma) = 1$.

3) STOCHASTIC OPTIMIZATION

To obtain $P_X(\mathbf{n})$, both the conditional probabilities and the process state probabilities must be computed. A node's conditional probability in (3) can be calculated as

$$P_X(\mathbf{n} \,|\, \sigma_i) = \frac{\delta(\sigma_i,\ \mathbf{n}) \cdot P_{\text{node}}(\mathbf{n})}{\sum_{\mathbf{m} \in \text{NodesFrom}(\mathcal{T},\ \sigma_i)} P_{\text{node}}(\mathbf{m})} \tag{4}$$

$$= \frac{\delta(\sigma_i,\ \mathbf{n}) \cdot P_{\text{node}}(\mathbf{n})}{\sum_{a \in \mathcal{A}.\text{FirstAt}(\sigma_i)} P(a)}, \tag{5}$$

where NodesFrom($\mathcal{T}$, σ) is the set of all nodes whose word can be emitted while in a state σ, where δ is a function defined as

$$\delta(\sigma,\ \mathbf{n}) = \begin{cases} 1 & \text{if } \mathbf{n} \in \text{NodesFrom}(\mathcal{T},\ \sigma) \\ 0 & \text{otherwise,} \end{cases} \tag{6}$$

and where $\mathcal{A}$.FirstAt(σ) is the set of symbols valid in the first position of the next emitted word. Assuming symbols are sorted in descending order of emission probability, $\mathcal{A}.\text{FirstAt}(\sigma_i) = \{a_j \in \mathcal{A} : j \geq i\}$ is the set of all but the first $i-1$ symbols in $\mathcal{A}$. Note that equation (5) holds by construction, because all nodes in NodesFrom($\mathcal{T}$, σ_i) are associated with words that start with $a \in \mathcal{A}.\text{FirstAt}(\sigma_i)$. The P_{node} sum of all nodes in any branch remains constant with node expansions as defined in Algorithm 3, hence symbol probabilities are unaffected by X's state in the second and subsequent positions of a word.

Continuing with P_X's calculation, X's state probabilities can be accurately computed assuming ergodicity of the source $\aleph$. Stationary Markov chain analysis is applied by considering the state transition probability matrix. The transition probability to σ_j given state σ_i can be expressed as

$$\mathbf{T}(\sigma_i,\ \sigma_j) = \sum_{\mathbf{n} \in \text{NodesTo}(\mathcal{T},\ \sigma_j)} P_X(\mathbf{n} \,|\, \sigma_i), \tag{7}$$

where

$$\text{NodesTo}(\mathcal{T},\ \sigma_j) = \{\mathbf{n} \in \mathcal{T} : |children(\mathbf{n})| = j - 1\} \tag{8}$$

is the set of nodes in $\mathcal{T}$ whose associated word's emission entails a transition to state σ_j. It is trivial to verify that $\mathbf{T}$ satisfies all necessary conditions for X to be a stationary Markov chain. In the proposed codec, π is defined as X's stationary state probability measure, which can be computed as shown in Algorithm 5: state probabilities are iteratively updated by multiplication with $\mathbf{T}$ until probability convergence of π. A small tolerance ε is allowed, e.g., $\varepsilon \approx 10^{-10}$, to avoid numerical instability. It was empirically found that Algorithm 5 converges within a few iterations.

The proposed method for generating individual, stochastically optimized code trees is given in Algorithm 4. Function MarkovOptimizedTree takes an alphabet $\mathcal{A}$, symbol probability P and a maximum number of nodes. A basic tree as described in Section IV-A.1 is generated for initialization purposes (line 3). The stationary state probabilities are computed using Algorithm 5 (line 6). These probabilities are then used to update the current tree with a new one using the routine defined in Section IV-A.2. If the newly generated code tree is not identical to the previous one, probabilities and trees are updated again. Otherwise, convergence is achieved,

[3] Note that the Greek letter π is hereinafter employed to denote *state* probability measures, as opposed to the Latin P used for symbols and words.

Algorithm 4 Generation of Code Trees Optimized Based on Stationary Markov Chain Analysis

function MarkovOptimizedTree($\mathcal{A}$, P, max_size)
  $t \leftarrow 0$
  $\mathcal{T} \leftarrow$ GetBasicTree($\mathcal{A}$, P, max_size)
  forever do
    $\pi \leftarrow$ TreeStationaryProbability($\mathcal{T}$)
    $X \leftarrow X_{\mathcal{T}}$
    $\mathcal{T}' \leftarrow$ GetProcessTree($\mathcal{A}$, P, Σ_X, π, max_size)
    if $\mathcal{T} = \mathcal{T}'$ **then**
      // Tree generation convergence
      return $\mathcal{T}$
    end if
    $\mathcal{T} \leftarrow \mathcal{T}'$
  end forever
end function

Algorithm 5 Calculation of the Stationary State Probabilities Associated With a Code Tree $\mathcal{T}$

function TreeStationaryProbability($\mathcal{T}$)
  $X \leftarrow X_{\mathcal{T}}$
  $\mathbf{T}(\sigma_i, \sigma_j) \leftarrow \sum_{\mathbf{n} \in \text{NodesTo}(\mathcal{T}, \sigma_j)} P_X(\mathbf{n} \mid \sigma_i)$
  $\pi(\sigma) \leftarrow \frac{1}{|\Sigma_X|} \quad \forall \sigma$ in Σ_X ▷ Uniform initialization
  forever do
    $\pi'(\sigma_i) \leftarrow \sum_{\sigma_j \in \Sigma_X} \pi(\sigma_j) \cdot \mathbf{T}(\sigma_j, \sigma_i) \quad \forall \sigma_i \in \Sigma_X$
    if $\sum_{\sigma \in \Sigma_X} \left(\pi(\sigma) - \pi'(\sigma)\right)^2 < \varepsilon$ **then**
      // State probability convergence
      return π
    end if
    $\pi \leftarrow \pi'$
  end forever
end function

i.e., the current code tree is optimized for sufficiently long input sequences. This iterative process was empirically found to converge in a few iterations.

B. OPTIMIZED CODE FORESTS

In Section IV-A, optimization is performed, assuming that a single tree is used to code all input symbols. In this section, multiple trees are jointly optimized assuming that different words can be emitted by different trees. Each emitted word determines the next tree to be used for coding. A *code forest* $\mathcal{F}$ is defined as a finite set of code trees, $\{\mathcal{T}_1, \ldots, \mathcal{T}_{|\mathcal{F}|}\}$, alongside the rules that dictate what tree within the set to use after emitting each word. Hereafter, $|\mathcal{F}|$ denotes the number of trees in the code forest. The general coding procedure for any code forest is provided in Algorithm 6.

In [47]'s algorithm, a code forest with exactly $|\mathcal{A}| - 1$ trees is produced. $\mathcal{T}_i$'s root has children for all but the $i-1$ most probable symbols. Therefore, $\mathcal{T}_i$ is designed specifically for state σ_i as defined in Section IV-A.2, where none of the $i-1$ most probable symbols may appear at the beginning of the next word. In [47], after emitting a node $\mathbf{n}$'s word, the next tree is the one designed for state $\sigma_{|children(\mathbf{n})|+1}$. In this section, we propose a more general forest design algorithm that produces a parametrizable number of trees. In contrast to [47], all tree roots of the proposed method have exactly $|\mathcal{A}|$ children so that any input sequence can be coded with any of these trees. Code trees in the forest $\mathcal{T} \in \mathcal{F}$ are optimized based on the transition rules of $\mathcal{F}$, as described below in Section IV-B.1. The number of trees determines the number of coding contexts that are effectively used for coding. In Section IV-B.2 a method is proposed to arrange word indices that allows highly competitive decompression throughput.

Algorithm 6 VF Coding of an Input Sequence $\mathbf{A}$ Using a Code Forest $\mathcal{F}$

function ForestCoding($\mathcal{F}$, $\mathbf{A}$)
  $\mathcal{T} \leftarrow \mathcal{F}_0$ ▷ $\mathcal{F}_0$: Arbitrary initial tree
  $\mathbf{n} \leftarrow \mathcal{T}.root$ ▷ $\mathbf{n}$: current node
  for each $a \in \mathbf{A}$ ▷ a: current input symbol
    if $\mathbf{n}$.HasEdge(a) **then**
      $\mathbf{n} \leftarrow \mathbf{n}$.Child($a$)
    else
      EmitWord($\mathbf{n}$.W) ▷ $\mathbf{n}$.W: $\mathbf{n}$'s word
      $\mathcal{T} \leftarrow \mathbf{n}.\mathcal{T}_{\text{next}}$ ▷ $\mathbf{n}.\mathcal{T}_{\text{next}}$: defined transition
      $\mathbf{n} \leftarrow \mathcal{T}.root$.Child($a$)
    end if
  end for
  if $\mathbf{n} \neq \mathcal{T}.root$ **then**
    FlushLastSymbols($\mathbf{n}$)
  end if
end function

1) STOCHASTIC FOREST OPTIMIZATION

In the proposed Algorithm 7, 2^Ω trees are produced, $\mathcal{F} = \{\mathcal{T}_1, \ldots, \mathcal{T}_{2^\Omega}\}$. Each tree contains exactly $|\mathcal{T}_i| = 2^K$ *Included* nodes. Parameters Ω and K are integers that must satisfy $K \geq \Omega \geq 0$, and $K \geq \lceil \log_2 |\mathcal{A}| \rceil$. For each tree in the forest, $2^{K-\Omega}$ of its words are defined so that $\mathcal{T}_1$ is used for coding after emitting them. The same number of words is devoted to the remaining trees in $\mathcal{F}$. For example, for $K = 8$ and $\Omega = 2$, 4 trees are created. Each tree contains 256 *Included* words, 64 of which define a transition to the first tree ($\mathcal{T}_1$), 64 to the second tree ($\mathcal{T}_2$), and so forth. As detailed later in Section IV-B.2, this design structure is imposed so that implementations can attain competitive HT performance.

A stochastic process Y is used to model the proposed forest's coding procedure. For each tree $\mathcal{T}$ in the forest, $|\mathcal{A}| - 1$ states are defined, $\{\sigma_1^{\mathcal{T}}, \ldots, \sigma_{|\mathcal{A}|-1}^{\mathcal{T}}\}$. The process Y is in state $\sigma_i^{\mathcal{T}}$ when tree $\mathcal{T}$ is selected to emit the next word $\mathbf{W}$, and none of the $i-1$ most probable symbols may appear at the beginning of $\mathbf{W}$. Therefore, $2^\Omega \cdot (|\mathcal{A}| - 1)$ states are defined in total for Y, $\Sigma_Y = \{\sigma_i^{\mathcal{T}}\}_{\mathcal{T} \in \mathcal{F},\, 1 \leq i \leq |\mathcal{A}|-1}$. The probability

Algorithm 7 Generation of Code Forests Optimized Based on Stationary Markov Chain Analysis

1: **function** MarkovOptimizedForest($\mathcal{A}$, P, K, Ω)
2: $\quad\mathcal{T}_\omega \leftarrow$ GetBasicTree($\mathcal{A}$, P, 2^K)
3: $\quad\quad\quad\quad\quad\quad\quad\quad\forall\omega \in \{1, \ldots, 2^\Omega\}$
4: $\quad\mathcal{F} \leftarrow \{\mathcal{T}_\omega\}_{\omega=1}^{2^\Omega}$
5: $\quad$DefineTransitions($\mathcal{F}$)
6: $\quad$**forever do**
7: $\quad\quad Y \leftarrow Y_\mathcal{F}$
8: $\quad\quad\pi \leftarrow$ ForestStationaryProbability($\mathcal{F}$)
9: $\quad\quad$**for each** $\omega \in \{1, \ldots, 2^\Omega\}$
10: $\quad\quad\quad\Sigma \leftarrow \{\sigma_i^\mathcal{T} \in \Sigma_Y : \mathcal{T} = \mathcal{T}_\omega\}$
11: $\quad\quad\quad\mathcal{T}'_\omega \leftarrow$GetProcessTree($\mathcal{A}$, P, Σ, π, 2^K)
12: $\quad\quad$**end for**
13: $\quad\quad\mathcal{F}' \leftarrow \{\mathcal{T}'_\omega\}_{\omega=1}^{2^\Omega}$
14: $\quad\quad$DefineTransitions($\mathcal{F}'$)
15: $\quad\quad$**if** $\mathcal{F} = \mathcal{F}'$ **then**
16: $\quad\quad\quad$**return** $\mathcal{F}$
17: $\quad\quad$**end if**
18: $\quad\quad\mathcal{F} \leftarrow \mathcal{F}'$
19: $\quad$**end forever**
20: **end function**

of a node $\mathbf{n}$'s word being emitted can then be expressed as a weighted sum of conditional probabilities,

$$P_Y(\mathbf{n}) = \sum_{\sigma_i^\mathcal{T} \in \Sigma_Y} P_Y\left(\mathbf{n} \middle| \sigma_i^\mathcal{T}\right) \cdot \pi\left(\sigma_i^\mathcal{T}\right), \tag{9}$$

where $\pi\left(\sigma_i^\mathcal{T}\right)$ is the probability of emitting a word while in state $\sigma_i^\mathcal{T}$. The conditional probability of a node $\mathbf{n}$ given $\sigma_i^\mathcal{T}$ is given by

$$P_Y\left(\mathbf{n} \middle| \sigma_i^\mathcal{T}\right) = \frac{\delta(\sigma_i^\mathcal{T}, \mathbf{n}) \cdot P_{\text{node}}(\mathbf{n})}{\sum_{\mathbf{n} \in \text{NodesFrom}(\mathcal{T}, \sigma_i^\mathcal{T})} P_{\text{node}}(\mathbf{n})} \tag{10}$$

$$= \frac{\delta(\sigma_i^\mathcal{T}, \mathbf{n}) \cdot P_{\text{node}}(\mathbf{n})}{\sum_{a \in \mathcal{A}.\text{FirstAt}(\sigma_i^\mathcal{T})} P(a)}. \tag{11}$$

As in Section IV-A, NodesFrom($\mathcal{T}$, $\sigma_i^\mathcal{T}$) denotes the set of nodes whose word can be emitted while in state $\sigma_i^\mathcal{T}$, $\mathcal{A}$.FirstAt($\sigma_i^\mathcal{T}$) is the set of all but the $i - 1$ most probable symbols in $\mathcal{A}$, and $\delta(\sigma_i^\mathcal{T}, \mathbf{n})$ is given by

$$\delta(\sigma_i^\mathcal{T}, \mathbf{n}) = \begin{cases} 1 & \textit{if } \mathbf{n} \in \text{NodesFrom}(\mathcal{T}, \sigma_i^\mathcal{T}) \\ 0 & \textit{otherwise}. \end{cases} \tag{12}$$

The state transition probability matrix of the forest's process Y can be expressed as

$$\mathbf{T}(\sigma_i^\mathcal{T}, \sigma_j^{\mathcal{T}'}) = \sum_{\mathbf{n} \in \text{NodesTo}(\mathcal{T}, \sigma_j^{\mathcal{T}'})} P_Y\left(\mathbf{n} \middle| \sigma_i^\mathcal{T}\right). \tag{13}$$

Here, NodesTo($\mathcal{T}$, $\sigma_j^{\mathcal{T}'}$) contains the nodes in $\mathcal{T}$ after which the process transitions to $\sigma_j^{\mathcal{T}'}$ and $\mathcal{T}'$ is used for the next word. Equation (11) holds by construction. Similar to Section IV-A, $\mathbf{T}$ defines a stationary Markov chain. Stationary state probabilities can be computed using the same iterative approach as in Algorithm 5. The new routine is hereinafter denoted ForestStationaryProbability($\mathcal{F}$). In this case, transitions between any two states are considered, including transitions between different trees. Therefore, the obtained probabilities well describe the long-run behavior of the coding process given a code forest.

Algorithm 7 is proposed to generate optimized code forest for an alphabet $\mathcal{A}$, symbol probability P, and parameters K and Ω. During initialization, 2^Ω trees are created, each with 2^K *Included* nodes and tree transitions as defined in DefineTransitions (lines 3 to 5). This routine sets those transitions with the restriction that $2^{K-\Omega}$ words from each tree in the forest must transition to any given $\mathcal{T} \in \mathcal{F}$. Since there are $2^{K+\Omega}$ total *Included* words in $\mathcal{F}$, $(K + \Omega)$-bit words are emitted. Therefore, any one-to-one mapping between the *Included* words and $0, \ldots, 2^{K+\Omega} - 1$ is valid. The proposed mapping used in DefineTransitions, which is optimized for highly efficient decompression, is detailed in Section IV-B.2. The remainder of Algorithm 7 alternates between stationary state probability updates (line 8), and the generation of 2^Ω trees and their transition rules (lines 9 to 14). Empirically, it was found that this iterative algorithm converges after a few minutes on a commercial desktop CPU for all tested parameters.

2) CONTEXT PARAMETRIZATION AND OVERLAPPED WORD INDICES

Each of the 2^Ω trees produced in an Algorithm 7's forest $\mathcal{F}$ can be considered a coding context, as they are selected based on the input symbols. As described in Section IV-B.1, trees are optimized based on their transitions as well as on the *Included* words' emission probability. DefineTransitions assigns which $2^{K-\Omega}$ words from each tree cause transition to each other $\mathcal{T} \in \mathcal{F}$. Finding optimal word-to-tree transitions is not trivial, so a heuristic is proposed as follows:

1) Nodes of each $\mathcal{T} \in \mathcal{F}$ are assigned to one of $|\mathcal{A}| - 1$ groups, denoted G_i. A node with $|children(\mathbf{n})|$ children is assigned to group $G_{|children(\mathbf{n})|+1}$. A list of *Included* nodes is produced.
2) Within each group G_i, *Included* nodes are sorted by P_{node}.
3) The sorted list of *Included* nodes is split into 2^Ω consecutive blocks of size $2^{K-\Omega}$. Words of the i-th block are defined to transition to $\mathcal{T}_i$.

This heuristic is designed so that trees $\mathcal{T}_i \in \mathcal{F}$ with lower indices are adapted to receiving transitions from more probable words. Moreover, the number of trees optimized based on the words in G_i depends on the number of words that transition to G_i. This results in more specialized trees available for the most probable cases, while less probable scenarios are still represented proportionally to an estimation of their likelihood.

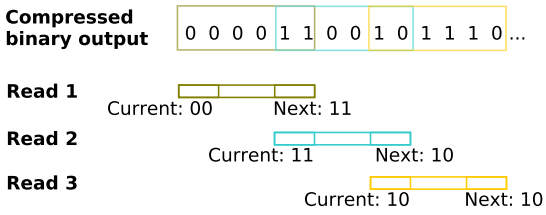

FIGURE 3. Diagram describing the reading of three overlapped words for $K = 4$ and $\Omega = 2$. In addition to the word index, the binary identifier of the current and next trees is obtained with every read operation.

The chosen transition structure enables a very efficient decoder implementation that removes the necessity for a memory access per emitted word to find the next tree to be used. More specifically, we propose to employ the Ω-bit suffix of each index's binary representation to identify which of the 2^{Ω} possible trees to use next. In addition, words are proposed to have $(K + \Omega)$-bit binary indices, of which the first Ω bits identify the tree to which a word's node belong. Therefore, with a single read operation of $K+\Omega$ bits, both the coded word and the next tree to be used are acquired. The next read starts K bits after the previous one. As a result, the last Ω bits of each word (identifying the next tree to use) are the first Ω bits of the next read of $K + \Omega$ bits. An example of the proposed overlapped word scheme is shown in Fig. 3 for $K = 4$ and $\Omega = 2$. It should be highlighted that the $K+\Omega$ bits read for each coded word contain information to reconstruct the intended list of symbols, as well as to signal the transition next tree to be used. Note that such an optimization is not possible with methods such as Yamamoto *et al.*'s, which require at least an additional memory access to determine the next tree to be used.

C. SELECTIVE INFORMATION CODING

The general motif of sections IV-A and IV-B is based on investing more resources –e.g., number of words within a tree's branch– to more probable events, in order to increase average word length and enhance compression ratio in those cases, to the detriment of less probable events. This motif can be generalized to further skew codec resources towards design aspects that can improve overall performance. The following improvements are proposed:

1) MODIFIED GOLOMB-RICE CODES

By considering the index $x \in \{0, \ldots, |\mathcal{A}| - 1\}$ of each input symbol in $\mathcal{A}$, r and q can be defined based on an integer parameter $S \geq 0$ as

$$r = x \mod 2^S \tag{14}$$

$$q = \left\lfloor \frac{x}{2^S} \right\rfloor. \tag{15}$$

Similar to Golomb-Rice codes, q and r are processed separately, and entropy coding is applied only to q. The binary expression of r corresponds to the S least significant bits of the input, which are output without further coding. Hereafter, S is referred to as the *shift* parameter. The quotients obtained in (15) are then compressed using a code forest, created as described in Section IV-B. The main advantage is that the effective alphabet size is divided by 2^S, and therefore it is easier to obtain longer average word lengths for a fixed dictionary size. For sources with r values distributed uniformly enough, gains due to longer word lengths are greater than the loss due to outputting r values without coding them.

2) UNREPRESENTED SYMBOLS

Lossless codecs must be able to represent any valid input, including those that contain highly improbable symbols. This is a source of inefficiency especially for large alphabets, because a non-negligible fraction of the words may be devoted to considering unlikely symbol events. Complementary to the modified Golomb-Rice method, another improvement to Algorithm 7 is proposed to address this issue. A symbol probability threshold Θ is selected, and the largest subset of symbols with probability sum below Θ are discarded. Algorithm 7 is used to produce an optimized code forest for the remaining symbols. The coding process is identical, except for the fact that when an unrepresented symbol is found, it is skipped. Instead, an auxiliary binary sequence is defined during initialization, and updated every time an unrepresented symbol is found, appending the unrepresented symbol's index in the input sequence, and its index in the original $\mathcal{A}$. For sufficiently low values of Θ, the number of such updates is small, hence it suffices to output those indices using fixed length binary representations. Empirically, we found Θ in the order of 10^{-6} to yield performance improvements for low entropy sources for which enough symbols can be discarded for that Θ.

V. PERFORMANCE EVALUATION

Given a codec $(\mathcal{E}, \mathcal{D})$ and an input $\mathbf{A} \in \mathcal{A}^+$, the *compressed data rate* is defined as

$$\mathcal{R}(\mathcal{E}, \mathbf{A}) = \frac{|\mathcal{E}(\mathbf{A})|}{|\mathbf{A}|}, \tag{16}$$

where $|\mathcal{E}(\mathbf{A})|$ is the length of the coded sequence in bits, $|\mathbf{A}|$ is the number of coded symbols, and $\mathcal{R}(\mathcal{E}, \mathbf{A})$ is expressed in bits per sample (bps). In this work, codec performance is evaluated based on three measures: compressed data rate, coding time, and decoding time. Section V-A discusses compressed data rate efficiency for synthetic memoryless sources and provides comparison with the state of the art in VF codecs. Compression rate for natural sources is evaluated jointly with coding and decoding time for the proposed codec and the best-performing publicly available HT codecs in Section V-B.

In order to attain full reproducibility of the performed experiments, except for the obtained time measurements, a code container is provided as supplementary materials for this manuscript. The interested reader is referred to this container for implementations of the tested codecs as well as of

the benchmark employed for generating the figures displayed in this document.

A. PERFORMANCE ON SYNTHETIC MEMORYLESS SOURCES

For a memoryless ergodic source $\aleph$ and long enough input sequences, Shannon proved that the lower bound for a codec's compressed data rate is the source's entropy [57],

$$H(\aleph) = -\sum_{a\in\mathcal{A}} P(a)\cdot\log_2 P(a)\,. \tag{17}$$

In this work, the *compression efficiency* of a codec $(\mathcal{E},\mathcal{D})$ for a given input $\mathbf{A}\in\mathcal{A}^+$ produced by $\aleph$ is defined as the ratio between that lower bound and the attained compressed data rate:

$$\eta_{\aleph}(\mathcal{E},\mathbf{A}) = \frac{H(\aleph)}{\mathcal{R}(\mathcal{E},\mathbf{A})}. \tag{18}$$

Synthetic pseudorandom sources are defined with $|\mathcal{A}| = 16$ and symbol probabilities based on a Laplacian distribution,

$$P(a_i) = \gamma\cdot\mathcal{L}(i\,|\,b,\,\mu=0) = \gamma\cdot\frac{e^{-i/b}}{2b}, \tag{19}$$

where γ is a constant defined so that $\sum_{a\in\mathcal{A}} P(a) = 1$, and b is a parameter that determines the resulting source entropy. For each source $\aleph$, an input sequence $\mathbf{A}_{\aleph}$ of length $|\mathbf{A}_{\aleph}| = 10^6$ symbols is generated so that symbol a appears $\max(1,[P(a)\cdot|\mathbf{A}_{\aleph}|])$ times. Symbols are then reordered in 16 pseudorandom ways to simulate the ergodicity and memoryless properties, The efficiency of a codec $(\mathcal{E},\mathcal{D})$ for a source $\aleph$'s generated input is defined as

$$\eta_{\aleph}(\mathcal{E},\aleph) = \eta_{\aleph}(\mathcal{E},\mathbf{A}_{\aleph})\,, \tag{20}$$

and average results for all trials are reported. Note average differences below 0.3% are observed between the trials.

1) CODE TREE EFFICIENCY

The stochastic optimization approach proposed in Algorithm 4 is compared to Algorithm 3 and to the *single tree* generation algorithm described by Yamamoto and Yokoo (YY) in [47]. Code trees are created for each algorithm with exactly $|\mathcal{T}| = 2^8$ nodes, and their compression efficiency is measured for $\aleph$ as described above. Efficiency results for these algorithms are shown in Fig. 4. Significant efficiency gains are observed for Algorithm 4, when compared to both Algorithm 3 and YY's code tree. Efficiency improvements are more noticeable for low entropy sources. This can be explained by the fact that the most probable symbol is rarely the first in a word for this type of sources and code trees. The proposed stochastic optimization correctly predicts this behavior and adapts code tree designs accordingly. Average compression efficiency gains due to stochastic optimization range from 0.08 to 0.11, depending on the parameter choice.

The effect of K on the proposed code trees is also studied. Fig. 5 shows efficiency results for the proposed method for

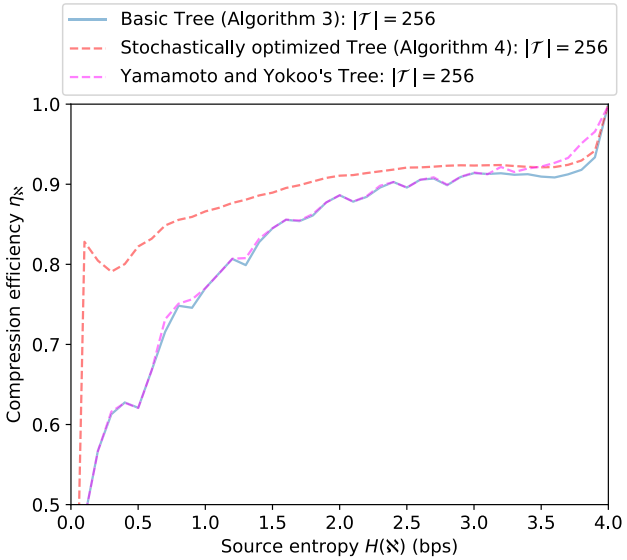

FIGURE 4. Compression efficiency for several code trees with identical number of words.

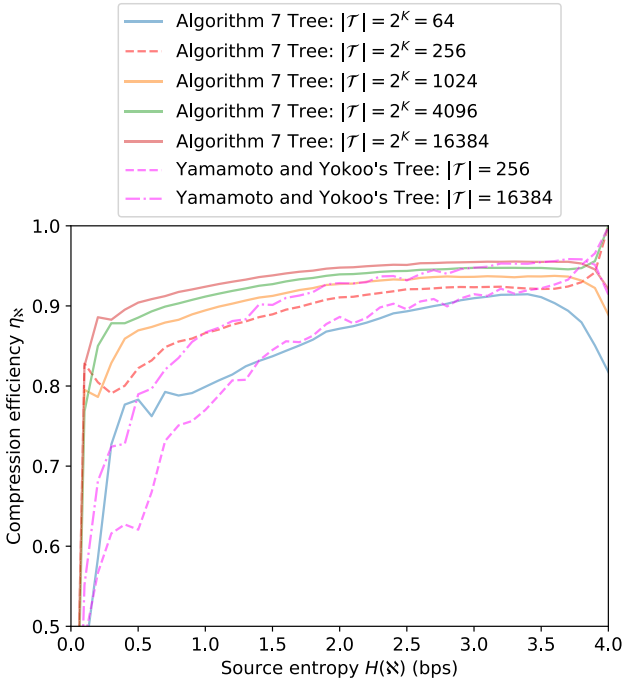

FIGURE 5. Compression efficiency for different code tree sizes.

K between 6 and 14, e.g., between 64 and 16384 *Included* words. As can be observed, better compression is observed for larger values of K. Notwithstanding, improvements due to increasing K become small when K is already large. For $|\mathcal{A}| = 16$, no significant efficiency gains are observed beyond $K = 14$. In practical applications, K should be ideally selected so that the code forest structure can be fit inside the L1 cache while providing competitive compression efficiency. Therefore, the optimal value of K depends on the alphabet size and the machine's cache size.

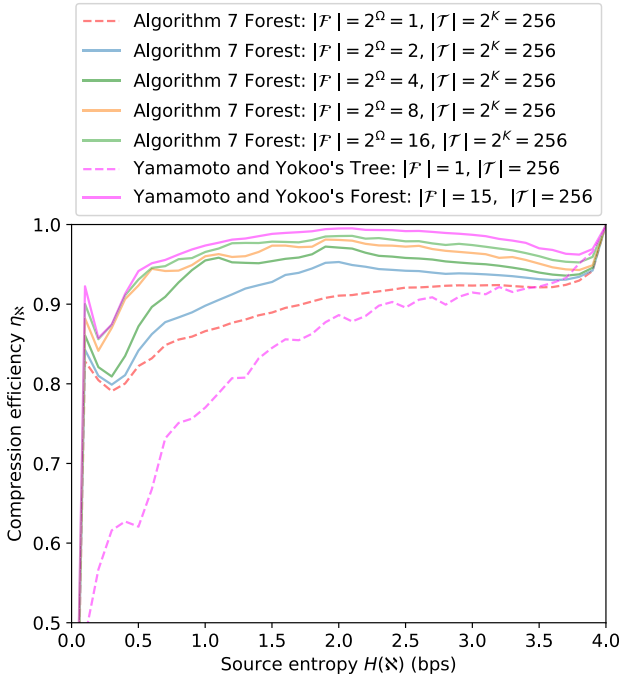

FIGURE 6. Compression efficiency for several code forests.

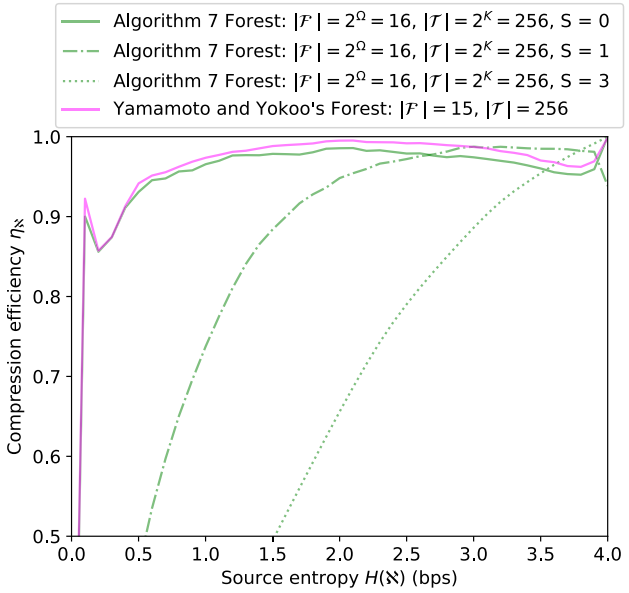

FIGURE 7. Impact of the modified Golomb-Rice shift parameter S on the compression efficiency of the proposed code forest.

TABLE 1. Test dataset properties.

Directory	Samples (millions)	Entropy (bps)
ISO [60]	264.24	7.40
Kodak [61]	28.31	7.65
Rawzor [62]	139.45	6.68
Mixed [63]–[65]	50.33	4.10
All	482.33	6.61

2) CODE FOREST EFFICIENCY

The code forest generation method proposed in Algorithm 7 is evaluated for different number of trees, $|\mathcal{F}| \in \{2^0, \ldots, 2^4\}$, each with exactly $|\mathcal{T}| = 2^8$ *Included* nodes. Note that for $|\mathcal{F}| = 1$, Algorithm 7 is equivalent to Algorithm 4. Compression efficiency for these forests, as well as for the ones obtained with YY's code tree and code forest generation algorithms for the same number of nodes per tree, is plotted in Fig. 6. Compression efficiency is consistently improved by using more than one tree in the code forest. In the proposed method, performance is increased with Ω, although increments beyond a certain point yield no significant gains. For $\Omega = 4$, the obtained compression efficiency curve is on average 0.95% worse than YY's code forest. It should be highlighted that YY's code forest structure does not admit the overlapped word structure described above, nor an equally efficient HT implementation. Note that the oscillatory nature of all performance curves is related to the intrinsic ability of variable-to-fixed codes to efficiently represent some symbol probability distributions. The interested reader is referred to [58], [59] for a more complete explanation to this behavior.

3) MODIFIED GOLOMB-RICE SHIFT

To study the impact of the modified Golomb-Rice shift parameter S in the proposed coded forest, compression efficiency is evaluated for $S \in \{0, 1, 3\}$ and for Yamamoto *et al.*'s forest generation algorithm. Results are shown in Fig. 7. Parameter values $S > 0$ reduce compression efficiency for medium and low entropies. This is as expected, since low entropy sources are more easily compressible, and therefore it is preferable to apply entropy coding to all bitplanes. On the other hand, for sources with high enough entropy, S enhances performance because the least significant bitplanes, which are hardly compressible, are output uncompressed. Therefore, more nodes can be devoted to encode more compressible parts of the input. Also as expected, the interception point between the unshifted ($S = 0$) and shifted ($S > 0$) plot lines is translated towards higher frequencies as S is increased. For the same reasons as before, performance efficiency for the maximum entropy depends on whether the total number of nodes, 2^K, is a power of symbols in the source alphabet after modified Golomb-Rice, i.e., $\lceil |\mathcal{A}|/2^S \rceil$.

B. PERFORMANCE ON NATURAL SOURCES

The proposed code forest generation algorithm is also tested on natural sources using data produced by real sensors of different types including image, scientific and textual. For these data, neither ergodicity nor the memoryless properties can be assumed, hence efficiencies higher than 1 are both theoretically possible and observed in practice. The following datasets containing visual contents have been included in the comparison: the RGB encoded ISO 12640-2 [60] standard color image set, the Kodak PhotoCD set [61], and the Rawzor [62] set. Data produced by other types of sources have also been included in the comparison. Namely, floating point data from chemical sensors [63], semi-structured text data describing medical product reviews and scores [64] and structured data from several physical dimensions taken at the Intel Berkeley Research lab [65] (grouped as *Mixed* in in Tables 1 and 2) are used in the comparison. This choice

TABLE 2. Average compression efficiency ($\eta_{\aleph}$), encoding speed ($\mathcal{E}_s$, 10^9·samples/s) and decoding speed ($\mathcal{D}_s$, 10^9·samples/s).

Codec	ISO			Kodak			Rawzor			Mixed			All		
	$\eta_{\aleph}$	$\mathcal{E}_s$	$\mathcal{D}_s$	$\eta_{\aleph}$	$\mathcal{E}_s$	$\mathcal{D}_s$	$\eta_{\aleph}$	$\mathcal{E}_s$	$\mathcal{D}_s$	$\eta_{\aleph}$	$\mathcal{E}_s$	$\mathcal{D}_s$	$\eta_{\aleph}$	$\mathcal{E}_s$	$\mathcal{D}_s$
Alg. 7 – K:10, Ω:0	1.38	0.13	2.28	1.13	0.12	1.72	1.06	0.15	3.20	0.65	0.12	3.76	1.11	0.14	2.93
Alg. 7 – K:10, Ω:2	1.40	0.12	2.16	1.14	0.11	1.78	1.08	0.14	2.67	0.66	0.10	2.18	1.12	0.13	2.41
Alg. 7 – K:8, Ω:0	1.37	0.18	2.60	1.13	0.16	2.09	1.06	0.23	3.67	0.65	0.15	3.97	1.10	0.20	3.32
Alg. 7 – K:8, Ω:2	1.40	0.17	2.46	1.15	0.15	2.10	1.08	0.19	2.87	0.66	0.14	3.36	1.12	0.17	2.78
FAPEC	1.13	13.97	16.51	1.05	12.28	14.26	1.06	5.36	6.00	0.58	8.04	8.77	1.02	8.63	9.92
FSE	1.44	0.25	0.43	1.14	0.22	0.35	1.11	0.23	0.85	0.69	0.26	0.39	1.15	0.24	0.64
Gipfeli	1.17	0.27	0.22	1.10	0.31	0.16	0.98	0.46	0.35	0.61	0.41	0.20	1.00	0.39	0.28
Huff0	1.46	0.25	0.42	1.19	0.21	0.40	1.12	0.22	0.46	0.70	0.26	0.44	1.17	0.24	0.44
LZ4	0.98	0.98	3.00	0.98	1.60	4.45	0.89	1.67	6.26	0.51	2.04	6.36	0.87	1.50	5.20
LZO	1.01	0.95	1.65	0.98	1.74	3.08	0.92	2.71	4.76	0.51	3.62	6.23	0.90	2.25	3.93
RLE	0.89	0.68	0.67	0.95	0.75	0.75	0.84	0.93	0.94	0.49	0.69	0.69	0.81	0.81	0.82
Rice	1.41	0.30	0.46	1.15	0.27	0.23	1.08	0.29	0.39	0.67	0.32	0.27	1.13	0.30	0.39
Snappy	1.00	0.97	2.07	0.98	1.83	3.62	0.91	2.60	5.43	0.51	3.52	6.22	0.89	2.19	4.43
Zstd	1.44	0.16	0.37	1.19	0.18	0.36	1.10	0.16	0.53	0.70	0.21	0.38	1.16	0.17	0.45

is intended to provide a representative, albeit non-exhaustive, selection of binary and text sources relevant to HT entropy codecs and exhibiting diverse properties. For compression, all files in these datasets are interpreted as one-dimensional sequences of 8-bit samples, i.e., $|\mathcal{A}| = 256$. In particular, image data is presented in band-sequential mode, components in raster order. The number of samples in each dataset as well as their average zero-order entropy is provided in Table 1. All input and output files are stored in an in-memory file system to decouple performance results from the type of long-term storage used.

Code forests produced by Algorithm 7 for parameters $K \in \{8, 10\}$, $\Omega \in \{0, 2\}$ are used to compress the aforementioned input sequences. Each sequence is divided in blocks of 4096 samples, and the zero-order entropy of each block determines which of 10 precomputed code forests to use for a given choice of K and Ω. This additional stage is included in the reported execution times, although it would not be needed when the sources' entropy is known a priori or otherwise calculated. Note that the optimal shift parameter S for each code forest is determined as a part of the precomputation. The proposed algorithm is compared with the best performing, publicly available HT entropy codecs. These include (in alphabetic order): FAPEC [25], FSE [23], Gipfeli [19], Huff0 [22], LZ4 [15], LZO [18], Snappy [16] and Zstd [20]. Rice and RLE were also implemented for this work for the sake of a more complete comparison. Note that compression algorithms specific for only one data type –e.g., images–, or with significantly higher computational complexity (e.g., JPEG-LS [66], JPEG 2000 [67], CALIC [68]), are out of the scope of this work and are not analyzed here.[4] To the best of the authors' knowledge, the tested algorithms are representative of the state of the art in HT coding/decoding. All compression and decompression routines in the tested algorithms are invoked with default parameters, except for FAPEC. For this method, the `-mt 1 -dtype 8 -np us` parameters are specified to guarantee that only one thread is employed, and that no decorrelation transformation is applied to the data before compression. This is to provide a fair comparison, since such transforms are out of the scope of this work and are not present in any of the other tested algorithms.

For each input sequence, entropy is obtained as described in (17), and used to calculate each codec's efficiency $\eta_{\aleph}$ as defined in Eq. (18). Compression and decompression speeds for each codec-sequence combination are obtained as the number of samples in that sequence, divided respectively by the encoding and decoding times, defined as the sum of measured CPU and System times on an Intel Xeon Platinum 8175M CPU at 2.50 GHz. To guarantee stability in the measurements, each codec-sequence combination is repeated until the accumulated computation time reaches a minimum of 1 s, and the average time is returned. All codecs are configured to use exactly one thread. Average efficiency results for each dataset are provided in Table 2. Average results for all input sequences are also included in Table 2, and plotted in Fig. 8.

1) COMPRESSION EFFICIENCY ($\eta_{\aleph}$)

The proposed code forests yield compression efficiency results comparable to the best codec for all tested datasets, i.e., Huff0, FSE and Zstd. On average, and depending on the choice of K and Ω, the proposed code forests yield compressed sizes 4.27% and 5.98% larger than Huff0. Consistent with previous discussion, using more than one tree in the forest ($\Omega > 0$) typically yields higher compression performance, with gains of up to 0.03. On the other hand, increasing the number of included nodes per tree, i.e., increasing the value of K beyond 8, enhances compression only if $\Omega = 0$. It should be highlighted that the observed efficiency differences between the proposed and the most efficient codecs is as expected, due to the additional design constraints imposed by VF codes as opposed to variable-to-variable codes such as Huff0, FSE and Zstd. Also note that for the ISO, Kodak and

[4]For instance, for the Rawzor image set, JPEG-LS compresses 5.2 times more slowly than the proposed method, while Kakadu JPEG 2000 and CALIC are, respectively, 9.7 and 18.4 times slower.

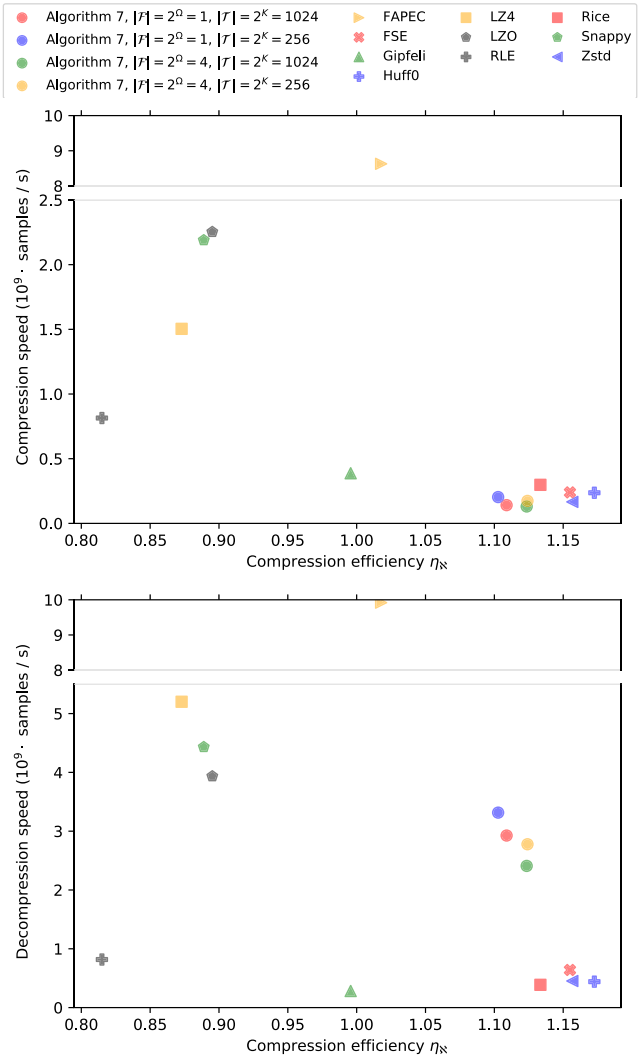

FIGURE 8. Average compression and decompression speeds as a function of compression efficiency.

Rawzor image datasets, most codecs attain efficiencies higher than 1. This is due to the fact that zero-order entropy (used in efficiency calculations) cannot detect any redundancy present among neighboring samples, whereas many of the tested codecs exploit it to obtain higher efficiency. In particular for the proposed algorithm, higher-order redundancy is exploited mainly via the shift parameter S, the use of several coding trees in the forest when $\Omega > 0$, and the per-block dictionary selection described above.

2) COMPRESSION AND DECOMPRESSION SPEED ($\mathcal{E}_s$, $\mathcal{D}_s$)

The proposed code forests exhibit compression speeds comparable to the algorithms with the highest compression efficiency. On average, our implementation of Algorithm 7 is able to encode $2.0 \cdot 10^8$ samples per second for $(K, \Omega) = (8, 0)$, i.e., 17.6% faster than Zstd and 16.7% slower than Huff0 and FSE. Golomb-rice codes yield good compression efficiency and speed, although their decoding speed is significantly lower than those of the proposed code forests. Other tested algorithms achieve a significantly higher sample throughput, most notably FAPEC, although their compression efficiency is not as high as that of the most efficient algorithms.

In terms of decompression speed, the proposed code forests yield a very competitive throughput, higher than those of the methods that yield the highest compression efficiency, i.e., Huff0, FSE and Zstd. This is one of the main goals of the proposed algorithm, based on variable-to-fixed codes. On average, the slowest of the tested code forests is 3.77 times faster than FSE, which in turn exhibits faster decompression than Huff0 and Zstd. Entropy codecs with lower compression efficiency (i.e., $\eta_{\aleph} < 1.0$ for the test sensor data) are able to decode faster than the proposed code forests. FAPEC dominates decompression speeds overall, in part due to the use of a heavily optimized, commercial implementation, which employs hardware-specific tuning such as single-instruction multiple-data (SIMD) assembly instructions.[5] In contrast, the proposed codec implementation is hardware agnostic. Note that the development of an optimized platform-specific version of the proposed codec is possible, but out of the scope of this paper.

For both compression and decompression, increasing K reduces attained speed. Even though the best value of K based on compression efficiency and execution speed depends on $\aleph$, empirically we found $K = 8$ to yield a favorable trade-off on the tested datasets. Increasing the shift parameter S also increases execution time, although very similar compression performance and execution times are observed for $K \in \{8, 10\}$ and $S > 0$.

VI. CONCLUSION

The design of HT entropy codecs with competitive performance is paramount to meet the ever increasing demands in terms of latency, bandwidth requirements, storage space usage and energy consumption. In many scenarios such as high-performance communication and computation systems, special importance is placed on decompression, since data are often decompressed multiple times or at critical points in the information pipeline. Notwithstanding, in most HT entropy codecs available, design decisions favor compression speed and efficiency, in detriment of decompression speed. The code forests proposed in this work are based on variable-to-fixed codes, which prioritize decompression speed while maintaining competitive compression efficiency. These forests are optimized using stationary Markov chain analysis, and achieve compression efficiency results within 1% of the state of the art in variable-to-fixed codes when tested on synthetic sources. In addition, the structure of the proposed codes enables highly efficient implementations that are not possible with other VF codes in the state of the art. When tested on natural sources against several paradigms of HT entropy codec, the proposed code forests

[5] Available at `https://www.dapcom.es/get-fapec/` at the time of writing.

attain compression efficiency competitive with the state of the art, with average decompression speeds 3.8 times faster than tested codecs with similar compression efficiency.

REFERENCES

[1] A. Beirami, M. Sardari, and F. Fekri, "Wireless network compression via memory-enabled overhearing helpers," *IEEE Trans. Wireless Commun.*, vol. 15, no. 1, pp. 176–190, Jan. 2016.
[2] M. Jägemar, S. Eldh, A. Ermedahl, and B. Lisper, "Automatic message compression with overload protection," *J. Syst. Softw.*, vol. 121, pp. 209–222, Nov. 2016.
[3] P. Ratanaworabhan, J. Ke, and M. Burtscher, "Fast lossless compression of scientific floating-point data," in *Proc. Data Compress. Conf. (DCC)*, Mar. 2006, pp. 133–142.
[4] M. Burtscher and P. Ratanaworabhan, "High throughput compression of double-precision floating-point data," in *Proc. Data Compress. Conf. (DCC)*, Mar. 2007, pp. 293–302.
[5] B. Nicolae, "High throughput data-compression for cloud storage," in *Data Management in Grid and Peer-to-Peer Systems*, A. Hameurlain, F. Morvan, and A. M. Tjoa, Eds. Berlin, Germany: Springer, 2010, pp. 1–12.
[6] D. Dong and J. Herbert, "Content-aware partial compression for big textual data analysis acceleration," in *Proc. IEEE 6th Int. Conf. Cloud Comput. Technol. Sci.*, Dec. 2014, pp. 320–325.
[7] Y. Liang and Y. Li, "An efficient and robust data compression algorithm in wireless sensor networks," *IEEE Commun. Lett.*, vol. 18, no. 3, pp. 439–442, Mar. 2014.
[8] A. Makhoul and H. Harb, "Data reduction in sensor networks: Performance evaluation in a real environment," *IEEE Embedded Syst. Lett.*, vol. 9, no. 4, pp. 101–104, Dec. 2017.
[9] S. Huang, J. Xu, R. Liu, and H. Liao, "A novel compression algorithm decision method for spark shuffle process," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2017, pp. 2931–2940.
[10] H. Harb and A. Makhoul, "Energy-efficient sensor data collection approach for industrial process monitoring," *IEEE Trans. Ind. Informat.*, vol. 14, no. 2, pp. 661–672, Feb. 2018.
[11] N.-M. Cheung, O. C. Au, M.-C. Kung, P. H. W. Wong, and C. H. Liu, "Highly parallel rate-distortion optimized intra-mode decision on multicore graphics processors," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 19, no. 11, pp. 1692–1703, Nov. 2009.
[12] C. Song, Y. Li, and B. Huang, "A GPU-accelerated wavelet decompression system with SPIHT and Reed–Solomon decoding for satellite images," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 683–690, Sep. 2011.
[13] F. Auli-Llinas, P. Enfedaque, J. C. Moure, and V. Sanchez, "Bitplane image coding with parallel coefficient processing," *IEEE Trans. Image Process.*, vol. 25, no. 1, pp. 209–219, Jan. 2016.
[14] P. Enfedaque, F. Auli-Llinas, and J. C. Moure, "GPU implementation of bitplane coding with parallel coefficient processing for high performance image compression," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 8, pp. 2272–2284, Aug. 2017.
[15] Y. Collet. (2011). *LZ4*. [Online]. Available: http://lz4.github.io/lz4
[16] Z. Tarantov and S. Gunderson. (2011). *Snappy*. [Online]. Available: http://www.google.github.io/snappy
[17] A. Farruggia, P. Ferragina, and R. Venturini, "Bicriteria data compression: Efficient and usable," in *Proc. Eur. Symp. Algorithms*. Berlin, Germany: Springer, 2014, pp. 406–417.
[18] M. Oberhumer. (1996). *LZO: Lempel Zip Oberhumer*. [Online]. Available: http://www.oberhumer.com/opensource/lzo
[19] R. Lenhardt and J. Alakuijala, "Gipfeli–high speed compression algorithm," in *Proc. Data Compress. Conf.*, Apr. 2012, pp. 109–118.
[20] Facebook. (2016). *Zstandard*. [Online]. Available: http://facebook.github.io/zstd/
[21] M. Martinez, M. Haurilet, R. Stiefelhagen, and J. Serra-Sagrista, "Marlin: A high throughput variable-to-fixed codec using plurally parsable dictionaries," in *Proc. Data Compress. Conf. (DCC)*, Apr. 2017, pp. 161–170.
[22] Y. Collet. (2013). *Huff0*. [Online]. Available: http://fastcompression.blogspot.de/p/huff0-range0-entropy-coders.html
[23] Y. Collet. (2013). *Finite State Entropy*. [Online]. Available: http://github.com/Cyan4973/FiniteStateEntropy
[24] J. de Vaan. (2007). *CharLS*. [Online]. Available: http://www.github.com/team-charls/charls
[25] J. Portell, R. Iudica, E. García-Berro, A. G. Villafranca, and G. Artigues, "FAPEC, a versatile and efficient data compressor for space missions," *Int. J. Remote Sens.*, vol. 39, no. 7, pp. 2022–2042, Apr. 2018.
[26] J. Bartrina-Rapesta, I. Blanes, F. Auli-Llinas, J. Serra-Sagrista, V. Sanchez, and M. W. Marcellin, "A lightweight contextual arithmetic coder for on-board remote sensing data compression," *IEEE Trans. Geosci. Remote Sens.*, vol. 55, no. 8, pp. 4825–4835, Aug. 2017.
[27] I. Blanes, A. Kiely, M. Hernández-Cabronero, and J. Serra-Sagristà, "Performance impact of parameter tuning on the CCSDS-123.0-B-2 low-complexity lossless and near-lossless multispectral and hyperspectral image compression standard," *Remote Sens.*, vol. 11, no. 11, p. 1390, 2019.
[28] M. Coatsworth, J. Tran, and A. Ferworn, "A hybrid lossless and lossy compression scheme for streaming RGB-D data in real time," in *Proc. IEEE Int. Symp. Saf., Secur., Rescue Robot.*, Oct. 2014, pp. 1–6.
[29] G. S. Martins, D. Portugal, and R. P. Rocha, "A comparison of general-purpose FOSS compression techniques for efficient communication in cooperative multi-robot tasks," in *Proc. 11th Int. Conf. Informat. Control, Autom. Robot. (ICINCO)*, vol. 2, 2014, pp. 136–147.
[30] M. Vecchio, R. Giaffreda, and F. Marcelloni, "Adaptive lossless entropy compressors for tiny IoT devices," *IEEE Trans. Wireless Commun.*, vol. 13, no. 2, pp. 1088–1100, Feb. 2014.
[31] C. J. Deepu, C.-H. Heng, and Y. Lian, "A hybrid data compression scheme for power reduction in wireless sensors for IoT," *IEEE Trans. Biomed. Circuits Syst.*, vol. 11, no. 2, pp. 245–254, Apr. 2017.
[32] D. Palma and R. Birkeland, "Enabling the Internet of arctic things with freely-drifting small-satellite swarms," *IEEE Access*, vol. 6, pp. 71435–71443, 2018.
[33] P. Deutsch, *DEFLATE Compressed Data Format Specification Version 1.3*, document RFC 1951, Netw. Work. Group, 1996, vol. 15.
[34] D. Harnik, E. Khaitzin, D. Sotnikov, and S. Taharlev, "A fast implementation of deflate," in *Proc. Data Compress. Conf.*, Mar. 2014, pp. 223–232.
[35] T. Srisooksai, K. Keamarungsi, P. Lamsrichan, and K. Araki, "Practical data compression in wireless sensor networks: A survey," *J. Netw. Comput. Appl.*, vol. 35, no. 1, pp. 37–59, Jan. 2012.
[36] T. Bicer, J. Yin, D. Chiu, G. Agrawal, and K. Schuchardt, "Integrating online compression to accelerate large-scale data analytics applications," in *Proc. IEEE 27th Int. Symp. Parallel Distrib. Process.*, May 2013, pp. 1205–1216.
[37] Y. Zhang, Y. Wu, and G. Yang, "Droplet: A distributed solution of data deduplication," in *Proc. ACM/IEEE 13th Int. Conf. Grid Comput.*, Sep. 2012, pp. 114–121.
[38] R. Filgueira, M. Atkinson, Y. Tanimura, and I. Kojima, "Applying selectively parallel i/o compression to parallel storage systems," in *Euro-Par Parallel Processing*, F. Silva, I. Dutra, and V. S. Costa, Eds. Cham, Switzerland: Springer, 2014, pp. 282–293.
[39] J. Janet, S. Balakrishnan, and E. R. Prasad, "Optimizing data movement within cloud environment using efficient compression techniques," in *Proc. Int. Conf. Inf. Commun. Embedded Syst. (ICICES)*, Feb. 2016, pp. 1–5.
[40] D. Abadi, S. Madden, and M. Ferreira, "Integrating compression and execution in column-oriented database systems," in *Proc. Int. Conf. Manage. Data (SIGMOD)*, 2006, pp. 671–682.
[41] B. P. Tunstall, "Synthesis of noiseless compression codes," Ph.D. dissertation, Georgia Inst. Technol., Atlanta, GA, USA, 1967.
[42] C. G. Boncelet, "Block arithmetic coding for source compression," *IEEE Trans. Inf. Theory*, vol. 39, no. 5, pp. 1546–1554, Sep. 1993.
[43] T. Raita and J. Teuhola, "Arithmetic coding into fixed-length codewords," *IEEE Trans. Inf. Theory*, vol. 40, no. 1, pp. 219–223, Jan. 1994.
[44] S. A. Savari, "Variable-to-fixed length codes and plurally parsable dictionaries," in *Proc. Inf. Theory Netw. Workshop*, 1999, pp. 453–462.
[45] M. D. Reavy and C. G. Boncelet, "An algorithm for compression of bilevel images," *IEEE Trans. Image Process.*, vol. 10, no. 5, pp. 669–676, May 2001.
[46] D.-Y. Chan, J.-F. Yang, and S.-Y. Chen, "Efficient connected-index finite-length arithmetic codes," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 11, no. 5, pp. 581–593, May 2001.
[47] H. Yamamoto and H. Yokoo, "Average-sense optimality and competitive optimality for almost instantaneous VF codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 6, pp. 2174–2184, Sep. 2001.
[48] Y. Xie, W. Wolf, and H. Lekatsas, "Code compression using variable-to-fixed coding based on arithmetic coding," in *Proc. Data Compress. Conf. (DCC)*, 2003, pp. 382–391.

[49] S. Yoshida and T. Kida, "An efficient algorithm for almost instantaneous VF code using multiplexed parse tree," in *Proc. Data Compress. Conf.*, 2010, pp. 219–228.
[50] S. Yoshida and T. Kida, "Analysis of multiplexed parse trees for almost instantaneous VF codes," in *Proc. IIAI Int. Conf. Adv. Appl. Informat.*, Sep. 2012, pp. 36–41.
[51] A. Al-Rababa'a and D. Dube, "Using bit recycling to reduce the redundancy in plurally parsable dictionaries," in *Proc. IEEE 14th Can. Workshop Inf. Theory (CWIT)*, Jul. 2015, pp. 62–65.
[52] F. Auli-Llinas, "Context-adaptive binary arithmetic coding with fixed-length codewords," *IEEE Trans. Multimedia*, vol. 17, no. 8, pp. 1385–1390, Aug. 2015.
[53] D. Dube and F. Haddad, "Individually optimal single- and multiple-tree almost instantaneous variable-to-fixed codes," in *Proc. IEEE Int. Symp. Inf. Theory (ISIT)*, Jun. 2018, pp. 2192–2196.
[54] M. Martinez, K. Sandfort, D. Dube, and J. Serra-Sagrista, "Improving Marlin's compression ratio with partially overlapping codewords," in *Proc. Data Compress. Conf.*, Mar. 2018, pp. 325–334.
[55] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Trans. Inf. Theory*, vol. IT-23, no. 3, pp. 337–343, May 1977.
[56] J. Duda, "Asymmetric numeral systems," 2009, *arXiv:0902.0271*. [Online]. Available: http://arxiv.org/abs/0902.0271
[57] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication*. Champaign, Il, USA: Univ. Illinois Press, 1949.
[58] O. Johnsen, "On the redundancy of binary Huffman codes (Corresp.)," *IEEE Trans. Inf. Theory*, vol. IT-26, no. 2, pp. 220–222, Mar. 1980.
[59] I. Blanes, M. Hernandez-Cabronero, J. Serra-Sagrista, and M. W. Marcellin, "Lower bounds on the redundancy of Huffman codes with known and unknown probabilities," *IEEE Access*, vol. 7, pp. 115857–115870, 2019.
[60] *Graphic Technology—Prepress Digital Data Exchange—Part 2: RGB Encoded Standard Colour Image Data*, Standard 12,640, International Standard Organization, 2004.
[61] R. Franzen. (1999). *Kodak Lossless True Color Image Suite: PhotoCD PCD0992*. [Online]. Available: http://r0k.us/graphics/kodak/
[62] S. Garg. (2011). *The New Test Images*. Accessed: Jun. 28, 2019. [Online]. Available: http://www.imagecompression.info/test_images
[63] A. Vergara, S. Vembu, T. Ayhan, M. A. Ryan, M. L. Homer, and R. Huerta, "Chemical gas sensor drift compensation using classifier ensembles," *Sens. Actuators B, Chem.*, vols. 166–167, pp. 320–329, May 2012.
[64] F. Gräßer, S. Kallumadi, H. Malberg, and S. Zaunseder, "Aspect-based sentiment analysis of drug reviews applying cross-domain and cross-data learning," in *Proc. Int. Conf. Digit. Health*. New York, NY, USA: ACM, Apr. 2018, pp. 121–125, doi: 10.1145/3194658.3194677.
[65] P. Bodik, W. Hong, C. Guestrin, S. Madden, M. Paskin, and R. Thibaux. (2004). *Intel Lab Data*. [Online]. Available: http://db.csail.mit.edu/labdata/labdata.html
[66] M. J. Weinberger, G. Seroussi, and G. Sapiro, "The LOCO-I lossless image compression algorithm: Principles and standardization into JPEG-LS," *IEEE Trans. Image Process.*, vol. 9, no. 8, pp. 1309–1324, Aug. 2000.
[67] D. S. Taubman and M. W. Marcellin, *JPEG 2000: Image Compression Fundamentals, Standards, and Practice*. Boston, MA, USA: Springer, 2012.
[68] X. Wu and N. Memon, "Context-based, adaptive, lossless image coding," *IEEE Trans. Commun.*, vol. 45, no. 4, pp. 437–444, Apr. 1997.

MANUEL MARTÍNEZ TORRES received the Ph. D. degree from the Karlsruhe Institute of Technology (KIT), Germany, in 2017.

He is currently a Postdoctoral with the Computer Vision for Human-Computer Interfaction Lab, KIT, where he focuses on applying computer vision to monitor sleep quality, and to develop assistive technologies for the visually impaired. He is passionate about teaching how to build complex systems that integrate multidisciplinary technologies.

Dr. Torres's practical course initiated, in 2011, received the best Practical Course Award at KIT, in 2016. In addition, he has contributed to the fields of object localization, deep learning, data compression, depth cameras among other sensing technologies. He has (co)authored over 25 articles and conference papers in the above domains, receiving the Best Poster Award.

MIGUEL HERNÁNDEZ-CABRONERO received the B.Sc. degrees in computer science and in mathematics from the Universidad Autónoma de Madrid, Madrid, Spain, in 2010, and the M.Sc. and Ph.D. degrees in computer science from the Universitat Autònoma de Barcelona, Spain, in 2011 and 2015, respectively.

He has held postdoctoral research positions at the University of Warwick, Coventry, U.K., and the University of Arizona, Tucson, AZ, USA. Since April 2019, he has been with the Universitat Autònoma de Barcelona. His research interests include data compression, especially medical and remote sensing image coding, and signal processing. He has coauthored multiple articles in these areas. He has also served as a reviewer for several journals and conferences in the field.

IAN BLANES (Senior Member, IEEE) received the B.S., M.S., and Ph.D. degrees in computer science from the Universitat Autònoma de Barcelona (UAB), Barcelona, Spain, in 2007, 2008, and 2010, respectively.

In 2010, he was a Visiting Postdoctoral Researcher with the Centre National d'Etudes Spatiales, Toulouse, France. Since 2011, he has been actively involved in the creation of new on-board data compression standards within the framework of the CCSDS Multispectral Hyper-Spectral Data Compression Working Group. Since 2003, he has also been with the Group on Interactive Coding of Images, UAB, where he is currently an Associate Professor. His research interest includes data compression in spaceborne instruments.

Dr. Blanes was the Second-Place Finisher in the 2007 Best Computer-Science Student Awards by the Spanish Ministry of Education.

JOAN SERRA-SAGRISTÀ (Senior Member, IEEE) received the Ph.D. degree in computer science from the Universitat Autònoma de Barcelona (UAB), Barcelona, Spain, in 1999.

From 1997 to 1998, he was with the University of Bonn, Bonn, Germany, funded by DAAD. He is currently a Full Professor with the Department of Information and Communications Engineering, UAB. He has coauthored more than 100 articles. His research interests include data compression, especially image coding for remote sensing applications.

Dr. Serra-Sagristà was a recipient of the Spanish Intensification Young Investigator Award, in 2006. He serves or has served as the Committee Chair for the Data Compression Conference. He serves or has served as an Associate Editor for the IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING and the IEEE TRANSACTIONS ON IMAGE PROCESSING.