IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 27, 2021, accepted July 22, 2021, date of publication July 27, 2021, date of current version August 5, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3100747

LOCO-ANS: An Optimization of JPEG-LS Using an Efficient and Low-Complexity Coder Based on ANS

TOBÍAS ALONSO, GUSTAVO SUTTER, (Member, IEEE), AND JORGE E. LÓPEZ DE VERGARA, (Senior Member, IEEE)
High Performance Computing and Networking Research Group, Escuela Politécnica Superior, Universidad Autónoma de Madrid, 28049 Madrid, Spain
Corresponding author: Tobías Alonso (tobias.alonso@uam.es)

This work was supported in part by the Spanish Research Agency through the Project AgileMon under Grant AEI PID2019-104451RB-C21.

ABSTRACT Near-lossless compression is a generalization of lossless compression, where the codec user is able to set the maximum absolute difference (the error tolerance) between the values of an original pixel and the decoded one. This enables higher compression ratios, while still allowing the control of the bounds of the quantization errors in the space domain. This feature makes them attractive for applications where a high degree of certainty is required. The JPEG-LS lossless and near-lossless image compression standard combines a good compression ratio with a low computational complexity, which makes it very suitable for scenarios with strong restrictions, common in embedded systems. However, our analysis shows great coding efficiency improvement potential, especially for lower entropy distributions, more common in near-lossless. In this work, we propose enhancements to the JPEG-LS standard, aimed at improving its coding efficiency at a low computational overhead, particularly for hardware implementations. The main contribution is a low complexity and efficient coder, based on Tabled Asymmetric Numeral Systems (tANS), well suited for a wide range of entropy sources and with simple hardware implementation. This coder enables further optimizations, resulting in great compression ratio improvements. When targeting photographic images, the proposed system is capable of achieving, in mean, 1.6%, 6%, and 37.6% better compression for error tolerances of 0, 1, and 10, respectively. Additional improvements are achieved increasing the context size and image tiling, obtaining 2.3% lower bpp for lossless compression. Our results also show that our proposal compares favorably against state-of-the-art codecs like JPEG-XL and WebP, particularly in near-lossless, where it achieves higher compression ratios with a faster coding speed.

INDEX TERMS Image codec, near-lossless compression, JPEG-LS, asymmetric numeral systems, low complexity, two-sized geometric distribution.

I. INTRODUCTION

There are scenarios in which, traditionally, lossless image codecs are used, among other reasons, due to the value of the information in the images (e.g. hard to obtain). Additionally, it may be required for legal reasons or to ensure system robustness, given that the level of uncertainty introduced by the quantization noise may not be admissible. However, within these scenarios, there are cases that allow for a coarser precision (depth of each pixel channel) with respect to the precision delivered by the sensor. In these cases, near-lossless codecs can be used to obtain higher compression ratios. This category of codecs is defined as a those that compress an image while allowing the user to set limits to the peak errors introduced in the decoded image, generally supporting lossless as the particular case where the error tolerance is set to 0.

There are many areas of applications that require this type of bounds on image quantization errors as in the case of image capturing satellites [1], [2] and medical imaging [3]–[5]. Within these applications, it is not uncommon to find examples that, additionally, have strong limitations on resources, energy consumption (sometimes indirectly because of limited dissipation capacity), latency and/or throughput [1], [4]–[6]. For this reason, low complexity is sought, understanding it

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

as the capacity of a system to run comparatively fast using limited resources.

Given the restrictions they face, many of these scenarios would benefit from or require compressing the images using custom hardware. This is the reason why an amenable hardware implementation is also a desired feature. Particularly, FPGAs are considered as an appropriate target, since the production volumes of most applications would not justify an ASIC. In addition, reconfigurability is a desired characteristic, allowing to update deployed systems. Moreover, image sensors can be connected directly to the FPGA, further improving the overall system performance.

Although in the state of the art it is possible to find codecs that have higher compression ratios, JPEG-LS is very well suited for these applications, given that it provides a competitive compression [7], [8] and, at the same time, permits high performance and low resource implementations, given their simple compute requirements and memory footprint. Because of this, several hardware implementations proposals have been published [1], [9]–[15], and it has even been used in NASA's Mars Exploration Rover mission [16].

Motivated by the existence of many applications that would benefit from low complexity lossless and near-lossless codecs, the observation of JPEG-LS optimization potential and the appearance of a new efficient and low complexity compression scheme, Asymmetric Numeral Systems (ANS), a series of modifications to the standard were developed, resulting in LOCO-ANS. At the cost of a low computational overhead, the proposed system achieves great compression ratio improvements.

The main contributions of this work are:

- An efficient and low complexity adaptive coder for sources with a geometrical distribution, which uses Tabled Asymmetric Numeral Systems (tANS) as the underlying technology [17], with an expected complexity similar to a Huffman coder but with efficiencies that closely approach to the model entropy [18], [19]. This coder is used as part of an adaptive system to code sources with a two-sided geometrical distribution.
- JPEG-LS codec is adapted to work with the proposed coder allowing a better compression, particularly, for lower entropy distributions, more common in near-lossless operation. The resulting system is capable of diverse trade-offs between resources and compression.
- From their conception, the proposed coder and modifications are hardware implementation oriented.
- The system prototype plus auxiliary code to create tables and run experiments are open sourced to the community [20].

The rest of this article is structured as follows: In section II a brief review of JPEG-LS and an introduction of ANS are presented. This is followed by an analysis of the optimization potential of JPEG-LS in section III and an overview of LOCO-ANS image encoder in section IV. Then, the coder and distribution parameters estimation details can be found in sections V and VI, while a methodology to select the coder configurations is provided in section VII. Next, the results of the experiments using the implemented prototype are presented in section VIII. Finally, in section IX the conclusions of this work are summarized. In appendix I, a table with the notation used is provided to make the equations easier to follow.

II. BACKGROUND

A. JPEG-LS

1) JPEG-LS BASELINE ALGORITHM

JPEG-LS was designed mainly for lossless compression with low complexity in mind and the objective to supersede the previous algorithms like the lossless mode of JPEG [21] and PNG [22]. Fig. 1 shows a high-level block diagram of the JPEG-LS encoder algorithm, which is based in LOCO-I [8], [23].

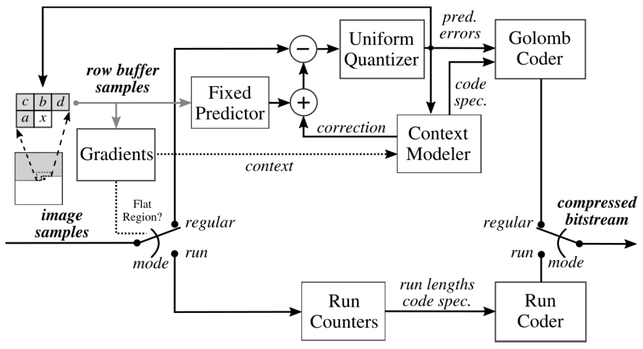

FIGURE 1. High-level JPEG-LS encoder block diagram. Source: Adapted from figure 1 of [23]).

It can be appreciated that it processes image samples using one of two modes, the regular and the run mode. In the regular mode, a prediction is computed and then corrected with an adaptive mechanism, resulting in a prediction error. This error is then quantized using a uniform mid-tread quantizer with a bin size $\delta = 2 * NEAR + 1$, where $NEAR$ is a parameter chosen by the user, which is equal to the maximum possible error of a pixel value in the decoded image. The quantized error is then coded by a low complexity adaptive block coder based on Golomb codes [24], which the authors call Golomb-power-of-2 (GPO2) codes.

As the GPO2 coder does not perform well when symbols come from a low entropy source, an adaptive run-length coder is used when smooth surfaces are detected by the gradients surrounding the current image sample. In the run mode, the run-length count is incremented when $|a - x| \leq NEAR$, where a is the pixel value when the count started and x is the new pixel. It is easy to see that, in both modes, lossless compression is obtained when $NEAR$ is set to 0.

To adapt the codes, contexts are used to keep prediction error statistics, which select coder parameters. These contexts are gradient defined. Gradients surrounding the new image sample are computed and then quantized separately, obtaining a vector of integers. The resulting vector is mapped to an identifier, which is used to access and update context statistics. In [23] a detailed description of the codec procedures can be found.

2) JPEG-LS EXTENSION

An extension of the standard [25] was proposed, based on LOCO-A (presented in [8]), changing the GPO2 and run-length coder used in LOCO-I by a single arithmetic coder and adapting the error distribution estimation procedures. These modifications closed most of the existing gap with CALIC [26], [27] at the cost of increasing the complexity of the system. This extension comes from the authors' recognition of the limitations of the original coder when dealing with low entropy distributions, as those that occur in near-lossless operation. In general, the higher error tolerance (parameter NEAR in JPEG-LS), the lower the entropy of the resulting quantized error distributions.

3) JPEG-LS HARDWARE IMPLEMENTATIONS

Several hardware architectures for JPEG-LS have been published [1], [9]–[15], however only a few are standard compliant. One of the main reasons for the lack of compliance is not supporting the run mode (which many deem as optional, although it is not [28]). In general, this is done to further simplify the hardware implementation. In [11] it was found complex to implement, while others noted the run mode is rarely used when losslessly compressing images coming from sensors, so decided not to implement it. Although this is generally true for lossless coding, long runs can arise, for example, when sensor saturation occurs. In satellite images, clouds tend to produce this effect. In the case of near-lossless operation, not supporting the run mode greatly impacts compression rates, as this mode is particularly important to complement the main weakness of prefix codes used in JPEG-LS when it comes to low entropy distributions (not able to produce an average code length below 1 bit for any symbol).

Another reason why the implementations did not adhere to the standard was the introduction of algorithm modifications to increase system throughput. Hardware implementations face mainly two bottlenecks: the context update, and the pixel quantization (and reconstruction) procedures. The latter only applies to near-lossless compression. Most implementations try to cope with these limitations by modifying the original algorithm and/or not supporting near-lossless compression (and thus avoiding the second bottleneck). In many cases, these modifications reduce the compression ratio.

Only two of the mentioned implementations support near-lossless compression [1], [9], but neither is standard compliant. In [1], several modifications are presented to the decorrelation and entropy coding stages, chiefly, the error tolerance (*NEAR* parameter) is modified within an image according to custom logic and only the GPO2 coder is employed, using a new adaptation algorithm. A close to standard compliant implementation is presented in [9], but it does not support the run-length coder. Although the performance of these two implementations cannot be directly compared, given the great difference in the technologies used in the experiments (0.22 μm process Xilinx XQR4062 in the former versus 40 nm process Xilinx Virtex 6 in the latter), the highest performing implementation supporting near lossless in the literature is the latter (51.68 Mpixels/second).

B. ASYMMETRIC NUMERAL SYSTEMS

Several years after the standardization of JPEG-LS and its extension, a new series of low complexity alternatives to arithmetic coding were proposed, Asymmetric numeral systems (ANS), initially introduced in [17], and later extended and compared to state-of-the-art compression algorithms, such as Huffman and arithmetic coding, in [18], [19].

ANS provides several possible algorithmic alternatives to implement coders. Particularly, tabled ANS (tANS) has the following properties:

- Suitability for high cardinality symbol sources.
- Capable of being used in adaptive coding settings.
- Able to match arithmetic coder [29] coding efficiencies (having an efficiency-memory resources trade-off).
- Moderate memory resource requirements.
- Has high-throughput implementations. For Field Programmable Gate Arrays (FPGA), encoder architectures were studied in [30] and decoder in [31], which can outperform Huffman decoding [32].

1) tANS OPERATION

From a black box perspective, tANS works as a Finite state machine (FSM) where the symbol to encode is the input and the current state is an integer, the ANS state, where ANS stores fractional bits of information. The output of the FSM ROM has the next state and the number of bits to take from the least significant part of the current state, which are then stored in the output bit file. From its design, tANS is meant to be implemented as a microcoded FSM (at least partially), and the FSM ROM is referred to as the *tANS table*. After a block of symbols is finished, the final state needs to be stored in the output bit file.

For the decodification, the binary bits are appended to the ANS state ($state \leftarrow (state \ll 1)\,|new\_bit$), until it is in a certain range (determined by the configuration of ANS). Then, this state is used to address the decoding table, obtaining the encoded symbol and the previous state. As implied by the decodification process, an ANS state is directly matched with a source symbol. Modifying the assignment of states to symbols changes the average number of bits ANS is going to generate for each of the source symbols.

tANS can be used to implement an adaptive coder given that switching to a different table changes the distribution ANS is tuned to. Using a particular table is referred to as an *ANS mode*. Of course, the decoder has to have the means to choose the same ANS mode that the encoder chose for each symbol. However, more attention has to be paid when using ANS in an adaptive manner, given that symbols are decoded in the opposite order they were coded (the last symbol coded is the first symbol that is decoded).

For more in depth explanation of the ANS algorithm and hardware implementations, refer to [18], [30], [31].

2) CODING EFFICIENCY

In general, the more bits used for the state, the more precisely the coder can be tuned to the desired distribution, which leads to a more efficient compression. Fig. 11 of [18] shows simulation results to understand the relationship between the number of ANS states used, the symbol alphabet size and the Kullback–Leibler divergence (KLD), also presenting the approximation $KLD \approx 0.5/(k)^2$ with $k = |S|/|A|$, where S is the set of states (in this work, it is generally assumed to be 2^{state_bits}), A is the set of symbols and $|\cdot|$ denotes the cardinality of the set. Eq. 1 summarizes our experience using the simple non-fine-tuning heuristic algorithm provided in the original work to create the tANS tables.

$$0.05/k^2 \lessapprox KLD_{tANS} \lessapprox 0.5/k^2 \qquad (1)$$

3) STATE SIZE AND MEMORY REQUIREMENT

Increasing precision comes at the cost of increasing memory requirements for the FSM ROM. However, the impact of this increment depends on the actual implementation and, as shown in [30], efficient architectures exist for large state configurations.

4) ANS STATE SIZE AND SMALL SYMBOL PROBABILITIES

In general, the tANS tables can be fine-tuned to obtain better results than the heuristic algorithm. However, there is a minimum symbol probability below which the table cannot be tuned to. In the case of 2-symbol sources, these tables are constructed by assigning to the higher probability symbol the first $2^{state\_bits} - 1$ states and the last state to the other symbol. These tables are referred to as *minimum entropy tables*.

Figure 2 shows the KLD achieved by 2-symbol minimum entropy tables as a function of the p Bernoulli distribution parameter for different ANS state sizes. Notice that 2-symbol sources would maximize the KLD for a given state size. From these curves, table 1 was obtained, which shows the minimum KLD achieved and at which p symbol probability.

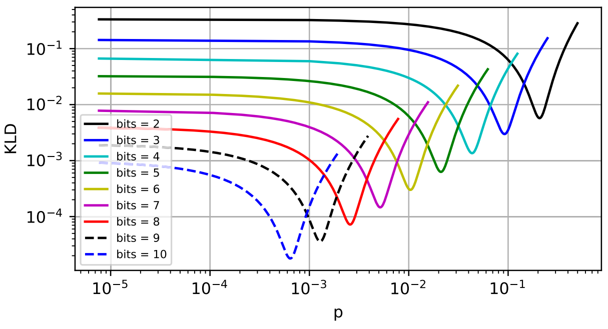

FIGURE 2. KL Divergence of 2-symbol tANS tuned to the minimum symbol probability as a function of the $P(0) = p$ probability for several ANS state bits.

TABLE 1. Performance of 2-symbol tANS tuned to the minimum symbol probability as a function of ANS state bits.

Address size	min KLD	$arg\,min_p\,KLD$	min prob/$(1/\lvert S\rvert)$
2	5.7273e-03	2.0750e-01	0.8300
3	2.9764e-03	9.2050e-02	0.7364
4	1.3537e-03	4.3425e-02	0.6948
5	6.2724e-04	2.1112e-02	0.6756
6	2.9950e-04	1.0406e-02	0.6660
7	1.4602e-04	5.1631e-03	0.6609
8	7.2063e-05	2.5766e-03	0.6596
9	3.5799e-05	1.2883e-03	0.6596
10	1.7833e-05	6.4114e-04	0.6565

This has to be taken into account when sizing the state size of the coder. For example, in an adaptive codification of a Bernoulli source, trying to tune tables to a p parameter equal or below the minimum probability observed in Table 1 will always result in the minimum entropy table. Then, adding these tables would not improve the coding efficiency, thus wasting resources.

C. TEST IMAGE DATASET

Throughout this work, the 8-bit gray image dataset maintained by Rawzor [33] was used to test the algorithms. A description of the images of the dataset can be found in Table 2, where the entropy was computed using a slightly modified version of the JPEG-LS baseline model (described in section III).

TABLE 2. Rawzor 8-bit gray dataset [33] description. Entropy estimation based on a modified version of the JPEG-LS baseline model (described in section III).

Image	Height x Width	Entropy	Type
artificial.pgm	2048 x 3072	0.7625	Synthetic
big_building.pgm	5412 x 7216	3.5437	Photographic
big_tree.pgm	4550 x 6088	3.6886	Photographic
bridge.pgm	4049 x 2749	4.1222	Photographic
cathedral.pgm	3008 x 2000	3.5343	Photographic
deer.pgm	2641 x 4043	4.6033	Photographic
fireworks.pgm	2352 x 3136	1.4218	Photographic
flower_foveon.pgm	1512 x 2268	1.9737	Photographic
hdr.pgm	2048 x 3072	2.1015	Photographic
leaves_iso_1600.pgm	2000 x 3008	4.4505	Photographic
leaves_iso_200.pgm	2000 x 3008	3.7648	Photographic
nightshot_iso_100.pgm	2352 x 3136	2.0217	Photographic
nightshot_iso_1600.pgm	2352 x 3136	3.9333	Photographic
spider_web.pgm	2848 x 4256	1.6722	Photographic
zone_plate.pgm	2000 x 3000	7.3368	Synthetic

III. JPEG-LS OPTIMIZATION POTENTIAL

The aim of this section is to establish a theoretical limit on improved compression due to the optimization of the prediction error coder for JPEG-LS given its statistical model.

A. THEORETICAL LIMIT OF CODER OPTIMIZATION

In order to understand the impact that a new coder could have, the average bits per pixel, bpp, obtained by JPEG-LS coder (using the implementation in [34], linked by jpeg.org web) was compared against the average symbol entropy using the statistical model employed in the standard to estimate the prediction error probabilities (coder symbols). As no implementations of the standard extension were found, only the baseline codec was included in the analysis.

Although the average symbol entropy would not take into account the effect of the compressed image header file size, this does not have noticeable impact, particularly for the image sizes of the used dataset. In JPEG-LS, the all 0 quantized gradients context, $\{0\}$, is handled differently as it is coded using the run-length coder, but in this analysis, given that the same statistical model is used for all contexts, it is treated as the rest.

In the standard, the error, ϵ, probabilities are estimated using a two-sided geometric distribution (TSG) as follows:

$$P(\theta, s)(\epsilon) = C(\theta, s)\theta^{|\epsilon - s|}, \epsilon = 0, \pm 1, \pm 2, \ldots, \tag{2}$$

where θ and s are the distribution parameters and $C(\theta, s) = (1-\theta)/(\theta^{1+s} + \theta^{-s})$ is a normalization factor. $\theta \in (0, 1)$ controls the rate of decay of the probabilities and $s \in (-1, 0]$ is the fractional bias (the sign of s is inverted compared to [23]).

In JPEG-LS baseline, s was decided to be in $(-1, 0]$ given that it was beneficial for their coding procedures. However, when computing the average symbol entropy, the bias cancellation procedure was configured so that $s \in (-0.5, 0.5]$, like in the standard extension. For this reason, the error sign flip applied when $s > 0$ was introduced, also employed in the standard extension. Additionally, the alternative model and estimators for the TSG proposed in [35] were used. This change does not imply a modification in the distribution but just a re-parametrization that simplifies the sequential parameter estimation. In this alternative model, each integer ϵ is mapped to a tuple (y, z), where:

$$y = y(\epsilon) \triangleq \begin{cases} 0, & \epsilon \geq 0 \\ 1, & \epsilon < 0 \end{cases} \tag{3}$$

and

$$z = z(\epsilon) \triangleq |\epsilon| - y(\epsilon) \tag{4}$$

Then, if $\epsilon \sim TSG(\theta, s)$, the variable $y \sim Bernulli(p)$ (where $p = (\theta^{1+s})/(\theta^{1-|s|} + \theta^{|s|})$) and the variable $z \sim Geometric(\theta)$ with the same θ as ϵ. For sample $t+1$, p is estimated (using $Beta(1/2, 1/2)$ as a prior) as follows:

$$\hat{p} = \frac{N_t + 1/2}{t+1}, \quad where \ N_t = \sum_{i=1}^{t} y_i \tag{5}$$

In [35] an optimal estimator of the probabilities of z_{t+1} is provided, however, the following estimator was used:

$$\hat{\theta} = \frac{S_t + \alpha}{S_t + t + \alpha + \beta} \tag{6}$$

where α and β are the parameters of the $Beta(\alpha, \beta)$ function used as a prior probability distribution. This last estimator, as noted by the authors of the model, is sub-optimal, but, in our experiments, it performed almost as well as the optimal one when using the same priors, with the advantage of being computationally simpler. To reflect the fact that as *NEAR* increases, θ decreases, $Beta(.5/(1 + \text{NEAR}/2), .5)$ was the prior used in the experiments.

The results can be seen in Table 3, where the column labeled as "Entropy_orig_ctx" was obtained using this model.

TABLE 3. JPEG-LS bpp vs TSG models estimated entropy.

Error	JPEG-LS	Entropy_orig_ctx	Entropy_fix_ctx
0	3.32	3.26 (1.7%)	3.26 (1.7%)
1	2.12	2.09 (1.1%)	2.01 (5.1%)
2	1.65	1.63 (1.4%)	1.51 (8.1%)
3	1.40	1.35 (3.2%)	1.24 (11.1%)
4	1.23	1.16 (5.4%)	1.06 (14.3%)
5	1.11	1.03 (6.8%)	0.92 (17.1%)
6	1.01	0.93 (7.9%)	0.81 (19.4%)
7	0.92	0.84 (8.5%)	0.72 (21.1%)
8	0.84	0.77 (8.3%)	0.65 (22.2%)
9	0.79	0.72 (8.4%)	0.6 (24%)
10	0.73	0.67 (9%)	0.55 (25.6%)

It can be seen that the larger the error tolerance, the less efficient JPEG-LS tends to be, having an inefficiency ranging from 1.7% for lossless compression to 9% for an error tolerance of 10.

B. OPTIMIZATION BY FIXING GRADIENT QUANTIZATION

In JPEG-LS, gradient quantization is a function of the NEAR parameter. As a result, the central quantization bin is expanded and the rest are scaled proportionally. Probably, the quantizer was designed in this manner to be able to use the run-length coder in this lower entropy scenario, but it was not considered necessary for a coder capable of handling low entropy distributions. For this reason, the quantization thresholds were fixed to those computed using NEAR = 0. As a result, the column labeled as "Entropy_fix_ctx" in Table 3 was obtained. As it can be seen in the table, this change would allow getting better compression ratios as the error tolerance increases. As expected, although this change reduced the estimated symbol entropy, it worsens the performance of JPEG-LS.

Additionally, a hardware implementation of the codec that supports multiple values of NEAR is slightly simplified resulting in smaller and faster logic for the gradient quantization.

It is worth noting that the changes introduced to the model, particularly to the gradient quantization, did not always result in an improvement in the estimated entropy. For example, the fixed gradient quantization worsens the entropy estimation of the synthetic image "zone plate" for all *NEAR* $>$ 0. However, the changes resulted in reduced entropy estimations in most cases, particularly for the photographic images, which are more relevant given the applications of low complexity lossless and near-lossless compression.

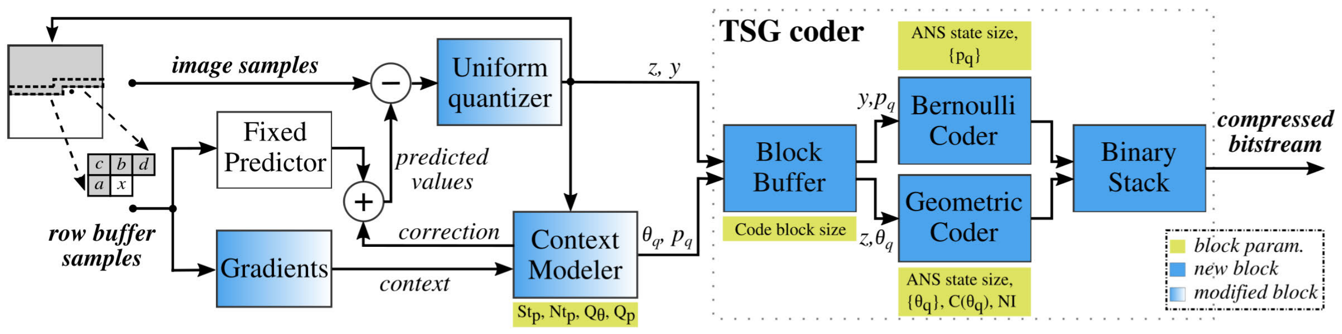

FIGURE 3. LOCO-ANS block diagram.

IV. LOCO-ANS OVERVIEW

LOCO-ANS block diagram can be seen in Fig. 3. It can be appreciated that the system has a single mode of operation and the coder was replaced with a new one (explained in section V). Additionally, some other modifications were introduced.

As it can be observed in the diagram, the alternative TSG model is used, then, z and y are computed and θ and p estimated (explained in section VI). Given the results obtained in section III, gradient quantization uses a fixed quantizer function (thresholds computed using $NEAR = 0$). Also, as in the standard extension, the prediction correction is configured so that the fractional bias, s, tends to stay in $(-0.5, 0.5]$ and the sign of the prediction error is inverted when $s > 0$.

A. ENCODING ALGORITHM SUMMARY

The encoder algorithm can be summarized as follows, where (*) denotes a new procedure and (†) one taken from the standard extension.

Scanning the image (with an *ibits* pixel depth) sequentially from left to right and from top to bottom:

1) Read the first pixel and store it directly, also updating the row buffer (*).
2) Read a new pixel.
3) Compute the gradients, quantize them (*) and obtain the pixel context.
4) Compute the fixed prediction.
5) Get the prediction bias and the TSG quantized parameters estimations, $\hat{p}_q$ and $\hat{\theta}_q$ for the context (*).
6) Correct the prediction using the bias and compute the prediction error.
7) Invert the sign of the prediction error if the context is negative and if $s > 0$ (†).
8) Obtain the quantized error using the NEAR parameter and reduce it modulo α, where $\alpha = 2^{ibits}$ if $NEAR = 0$ else $\alpha = (2^{ibits} - 1 + 2 * NEAR)/(2 * NEAR + 1)$.
9) Compute z and y and store it in the coder input buffer with their distribution parameters (*).
10) Check if the symbol block is complete, and if so, use the coder presented in section V to process the whole block and append the resulting binary stack to output bit stream (*).
11) Reconstruct the pixel and store it in the row buffer.
12) Update the prediction bias (†) and the TSG parameters estimations (*).
13) If there are more pixels in the image, return to step 2.

Although presented as an ordered list, notice that some of these steps can be done completely or partially in parallel.

V. AN ANS-BASED CODER FOR TSG SOURCES

To use tANS in an adaptive setting, in general, one table per symbol distribution is required, so there is a trade-off between table resources and KLD. Additionally, more tables can also imply a reduction in the coder throughput.

Given the simplicity of the parameter estimation procedures and the coding efficiency of ANS, the proposed system encodes the (y, z) tuple components separately, instead of the TSG distributed error. Notice that choosing to code the (y, z) tuple components independently, allows having tables tuned to the distributions of each component instead of tuned to the tuple join distributions, which is needed if using the TSG model described by eq. 2. In this way, the number of required tables is equal to $|\{\hat{\theta}_q\}| + |\{\hat{p}_q\}|$ instead of $|\{\hat{\theta}_q\}| * |\{\hat{p}_q\}|$ tables, where $|\{\hat{\theta}_q\}|$ and $|\{\hat{p}_q\}|$ are the number of reconstruction values supported for θ and p, respectively.

In this section, the codification procedures for y and z variables are presented. To simplify the explanation, as most algorithms do not strictly depend on ANS, it is first assumed that the encode and decode order are the same (which is not true for ANS), addressing the codification order required by ANS in section V-C;

A. ADAPTIVE BERNOULLI CODER

Coding the y binary variable with tANS is simple. Given a quantized estimation of the Bernoulli parameter $\hat{p}_q = Q_p(\hat{p})$, where Q_p is the chosen quantization function for the p parameter, a unique index is assigned to it, which is used to select the ANS table tuned to $\hat{p}_q$.

To half the number of required tables, if $\hat{p}_q > 0.5$, then y is inverted and $\hat{p}_q$ is set to $1 - \hat{p}_q$. On the decoder side, when $\hat{p}_q > 0.5$, then $\hat{p}_q \leftarrow 1 - \hat{p}_q$ is used to select the decode table, and the obtained symbol is inverted.

Note that assuming the TSG distribution hypothesis holds and that the bias cancellation procedure works well, and given that $p = (\theta^{1+s})/(\theta^{1-|s|} + \theta^{|s|})$ and $s \in (-0.5, 0.5]$ then $p \in [\theta/2, 0.5]$. In practice, using the Rawzor dataset, limiting the $\hat{p}$ to that range does not increase the bpp, except for the "zone plate" artificial image.

B. BASIC GEOMETRIC CODER

Given a symbol z coming from an infinite alphabet source with a geometric distribution and a quantized parameter estimation $\hat{\theta}_q = Q_\theta(\hat{\theta})$, where Q_θ is the chosen quantization for the θ parameter, the probabilities of z are computed as:

$$P_{(\hat{\theta}_q)}(z) = (1 - \hat{\theta}_q) \cdot \hat{\theta}_q^{\,z} \qquad (7)$$

To code this type of symbol source using tANS, one main challenge had to be overcome. Taking into account the maximum possible value of z for the image compressing application, the cardinality of the symbol source is very large, which leads to high resource requirements for the tANS tables. Additionally, eq. 7 shows that the probabilities of z can decrease very fast. So, as seen in section II-B4, ANS would require an impossibly large state to cover the whole z range, which, in turn, exponentially increases the memory requirements. This could be addressed with binarization, but there is an alternative enabled by the memoryless property of geometric distribution, which allows a simpler, scalable and generally higher throughput system.

Both the large cardinality and high probability precision problems can be addressed by using conditional probabilities when the symbol z is larger than an implementation defined threshold. Symbols in the range $[0..(C - 1)]$ are coded directly, choosing the ANS mode (ANS table computed for a certain distribution) according to the provided $\hat{\theta}_q$, which also determines the C constant. For larger symbols, the coder inserts C, which stands for "$z \geq C$". Applying the memoryless property, it can be seen that the distribution of $(z-C)$ given that $z \geq C$ is the same as z. For this to be strictly true, z should come from an infinite set not a constrained one, as in the case of error residuals, but the set is large enough, so there is no significant difference, at least, for the $\hat{\theta}$ seen in practice. Then, using the same ANS mode (as they have the same distribution), the system tries to code $(z-C)$ and, again, if it is greater or equal to C, it inserts C. This process is repeated until a symbol different that C is coded.

Notice that in this way, without the need of deriving probabilities for the decomposed symbols or any additional statistics gathering process, and using a stateless coder with $C+1$ symbols, any number originated from an infinite alphabet source with the memoryless property can be optimally encoded. In this way, for each supported $\hat{\theta}_q$, just one tANS table tuned to a $C + 1$ symbol source is required.

C. CODIFICATION ORDER FOR ANS

If ANS is used to code the symbols, then for the bitstream to be decodable, the codification order must be inverted.

1) SYMBOL BLOCK CODIFICATION ORDER

As mentioned before, the ANS output binary acts as a Last In, First Out (LIFO) memory, so prior to coding, the symbols are stored with the necessary adaptation parameters and coded in reverse order, as proposed in [18]. In this case, $\hat{\theta}_q$ and $\hat{p}_q$ parameters should be stored alongside the (y, z) tuple. In some cases, it is not possible or desirable (added latency) to store these variables for the whole image, so smaller blocks can be used at the cost of some additional bits (the final ANS state needs to be sent after each block and small inefficiencies can arise due to word alignment) effecting slightly the overall coding efficiency. The decoder needs to know the block size, which can be included in the compressed image header.

In general, the additional bits per symbol due to the need of transmitting the final ANS state at the end of each block and the requirement of aligning a new block to a certain word size is going to be, on average:

$$\overline{KLD} = (state\_bits + (word\_bits - 1)/2)/block\_size \qquad (8)$$

For example, for a 6 bit ANS state, aligning binary blocks to bytes and using a block size of 2048 pixels, $\overline{KLD} = (6 + (8 - 1)/2)/2048 \; = 0.0046 \; bits/pixel$.

As suggested in [18], the initial state of the ANS coder can be used to carry information, but, in the system prototypes, the initial ANS state is used as a sanity check of each block. That is, the encoder always sets the initial state to 0 (actually to 2^{state_bits}), and the decoder checks after each block that the final ANS state is 0 (corresponding to the first in the encoder side).

In hardware implementations, to avoid stalls, a ping-pong buffer should be used. In this manner, a block can be processed, while the next one is being generated.

2) SUBSYMBOL CODIFICATION ORDER

If y is coded before z, then z is decoded before y. Additionally, for each symbol z, the order of operations described in section V-B is as the decoder would see them. The encoder should proceed in the reverse order, inserting first the last subsymbol the decoder should see. It is not hard to see that the value of that subsymbol is $z \mod C$ (trivial to implement if C is chosen to be a power of 2). After, if required, it inserts a sequence of n C sub-symbols, where $n = (z - (z \mod C))/C$.

Finally, the codification of a single z symbol could be implemented as seen in Fig. 4. There, *store_in_binary_stack* function call deals directly with the output binary and its arguments are an integer variable with the bits to store and the number of bits to take starting from the least significant bits. The ANS tables are stored in the array *ANS_table*, which is addressed by the quantized distribution parameter id, the current state of the ANS coder and the new symbol to encode. Each element of the table is a structure with the number of bits that should be sent to the output and the next ANS state. Note that ANS operation can be implemented differently [19], [30].

```
Require: $z$
Require: $param$
 1: $c \leftarrow get\_cardinality(param)$
 2: $remaining\_sym \leftarrow z$
 3: $subsym \leftarrow z \mod c$
 4: repeat
 5:     $remaining\_sym \leftarrow remaining\_sym - subsym$
        {tANS coding}
 6:     $obits \leftarrow ANS\_table[param][state][subsym].bits$
 7:     store_in_binary_stack($state, obits$)
 8:     $state \leftarrow ANS\_table[param][state][subsym].nx\_st$
 9:     $subsym \leftarrow c$
10: until $remaining\_sym = 0$
```

FIGURE 4. **Codification procedure for a single geometrically distributed symbol using tANS.**

3) BINARY STORE ORDER

As the decoder reads the binary bits in the inverse order, the encoder generates them and to avoid the need of appending a header to each binary block, bits should be stored in the reverse order as they are produced. This can be easily implemented storing coder output bits in a stack, and then copying the whole binary block below the previous binary block.

D. GEOMETRIC CODER ITERATIONS

Although this algorithm may appear to be slow for its iterative nature, even with small $C \in [1, 16]$ and for the θ observed in 8-bit images, it is not. This can be appreciated in Fig. 5, where the expected iterations per symbol ($\bar{i}$) were plotted, which is computed as follows:

$$\bar{i} = \frac{1}{1-\theta^C} \tag{9}$$

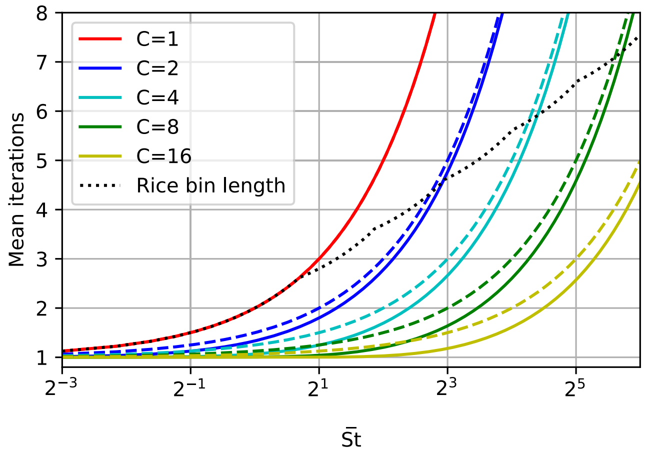

FIGURE 5. **Geometric coder mean iterations as a function of $\overline{S_t}$ compared to number of iterations resulting from a Rice-based binarization strategy. The approximations are shown with dashed lines.**

Note that the equation can be approximated with $\bar{i} \approx \overline{S_t}/C + 1$ for high $\overline{S_t}/C$, where $\overline{S_t} = \sum z_i/t = \theta/(1-\theta)$.

As a reference, Fig. 6 shows the distribution of $\overline{S_t}$ for different values of *NEAR* for the Rawzor dataset.

Notice that the larger $\overline{S_t}$, the smaller rate at which $P(z)$ decreases. Additionally, C can be a function of $\hat{\theta}_q$. So, in general, although $\bar{i}$ increases almost linearly with $\overline{S_t}$, the maximum C value for a given an ANS state size tends to increase with $\overline{S_t}$. Of course, the state size has to be large enough to be able to code θ and $(1-\theta)$ (the two symbols for $C = 1$).

An alternative strategy to the one presented in Fig. 4 would be to binarize the symbol and then proceed with a binary coder. Instead of a trivial binarization, this procedure could consist in using Rice-codes [36] for the symbol, which is a similar method to the one employed in the JPEG-LS standard extension. Fig. 5 allows to compare the iterations required by the proposed method with the average number of bits (and thus iterations of the binary coder) resulting from a Rice coding binarization strategy. There, the k rice parameter was chosen as the closest integer to $-log_2(-log_2(\theta))$. It can be seen that in range of interest, with small values of C, the proposed method requires fewer iterations. Moreover, no binarization or bit probability modeling is required.

Although the coder can be configured so that $\bar{i}$ stays within some desired bounds, the maximum possible iterations are higher, which can lead to data loss if buffers are not correctly sized. Given that there are many situations in which buffer sizes and/or latency are highly constrained, this issue is addressed in the next section.

E. LIMITATION TO CODER ITERATIONS AND SYMBOL EXPANSION

One of the concerns that arises when analyzing the proposed coding algorithm is the possibility of bursts of symbols requiring many cycles to code them and, particularly for the smaller θ, the possibility of local expansion. Both burst of long iterations and expansions have to be considered when sizing buffers before and after the coder, respectively. For this reason, it would be desirable to have a direct mechanism to limit them.

A way of limiting both the expansion and the number of iterations would be the following: The maximum number of iterations (NI) is chosen. Then, NI consecutive sub-symbols C will act as an escape mechanism, after which z is stored directly using $z\_bits = \lceil log_2(max(z)+1) \rceil$. Given the modulo reduction applied to the prediction error in JPEG-LS, z can be coded with $(ibits - 1)$ bits, where $ibits$ is the pixel depth of the input image.

Alternatively, the residual $(z - NI \cdot C)$ could be stored, which in some configurations might require fewer bits to code. What is more, the GPO2 coder could be used to code this residual, selecting k from a small array indexed by the distribution parameter. These codes are more efficient for high θ geometric distributions, which at the same time are the most likely to require this mechanism for a given C.

For an ANS implementation of the coder, the order is reversed. If $z \geq NI \times C$, then z (or the residual) is coded

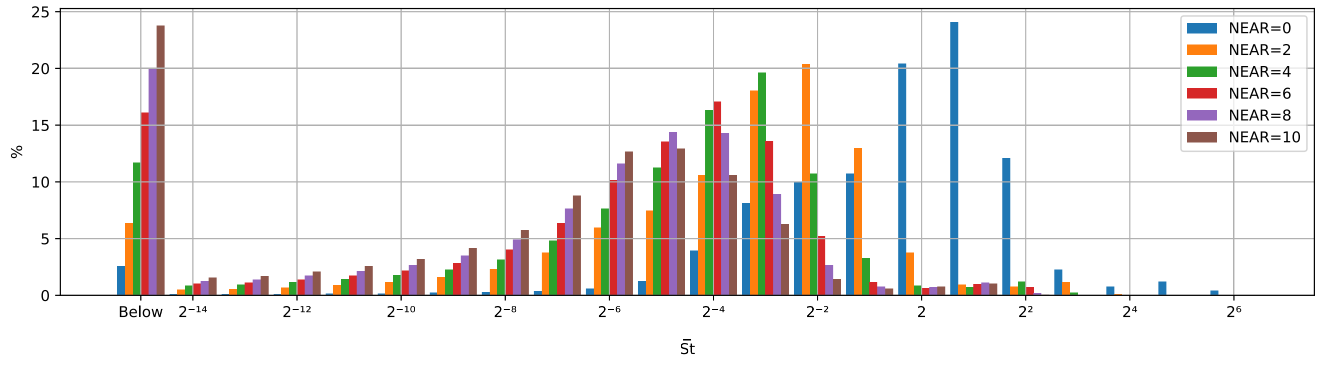

FIGURE 6. $\overline{S_t}$ **Histogram for Rawzor dataset using different NEAR parameters. Bins bounds are placed at** $(2^x, 2^{x+1})$ **with** 2^x **representing the bin.**

```
Require: $z$
Require: $param$
Require: $z\_bits$
 1: $c \leftarrow get\_cardinality(param)$
 2: $remaining\_sym \leftarrow z$
 3: $subsym \leftarrow mod(z, c)$
 4: if $z \geq NI * c$ then
 5:     store_in_binary$(z, z\_bits)$
 6:     $remaining\_sym \leftarrow NI * c$
 7:     $subsym \leftarrow c$
 8: end if
    {code $z$ or escape symbol with tANS}
 9: repeat
10:     $remaining\_sym \leftarrow remaining\_sym - subsym$
        {tANS coding}
11:     $obits \leftarrow ANS\_table[param][state][subsym].bits$
12:     store_in_binary_stack$(state, obits)$
13:     $state \leftarrow ANS\_table[param][state][subsym].nx\_st$
14:     $subsym \leftarrow c$
15: until $remaining\_sym = 0$
```

FIGURE 7. **Codification of single** z **limiting the iterations.**

and, after that, NI consecutive C sub-symbols are coded using the ANS encoder as usual. On the receiver side, upon seeing NI C sub-symbols, it will get out of the loop and proceed to get z directly (or through the Golomb decoder). Note that, as before, if C is chosen to be a power of 2, $NI \times C$ is just a binary shift. Alternatively, those values can be stored, and then, retrieved using the parameter distribution identifier. Finally, for the simpler case where the GPO2 coder is not used to code the residual, the algorithm can be expressed as in Fig. 7, where a new input is required, z_bits.

1) IMPLICATIONS ON CODING EFFICIENCY

The implementation of this iteration limitation mechanism will tend to decrease the coding efficiency of the coder. In its simpler version, it forces all symbols equal or above $NI \times C$ to be coded using a fixed amount of bits (z_bits). The KLD

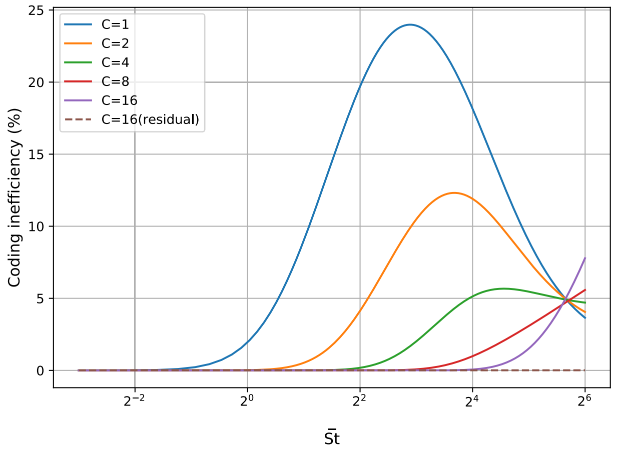

FIGURE 8. **Coding inefficiency (*KLD*/*Entropy*) caused by the iteration limitation mechanism using the direct** z **codification after the escape mechanism. Curves for** $NI = 7$.

can be obtained as:

$$KLD(L, \theta) = \theta^L \cdot \left(z\_bits - Entropy(z|z \leq max(residual)\right) \tag{10}$$

where $L = NI \cdot C$.

The code inefficiency ($KLD/Entropy$) due to the use of the simple coding of z or its residual can be observed in Fig. 8, setting NI to 7 and using small values of C, for $\overline{S_t}$ in the range observed in dataset. As it can be seen, KLD can be relatively small even for the simplest codification and using small numbers of C.

2) UPPER BOUND ON THE CODE LENGTH

As mentioned before, apart from limiting the iterations, it would be useful to obtain an upper limit on the code length. For each $\hat{\theta}_q$, if $Max(z) \geq NI \times C$, there are two symbols that could have the maximum code length. These are either $z \geq NI \times C$ (all symbols in this set are coded with the same length) or $z = NI \times C - 1$. Then, if $NI > 0$, an upper limit for a single symbol and a given $\hat{\theta}_q$ would be:

$$\begin{aligned} \textit{Max code length} \leq \; & \max_{state}(tANS_{\hat{\theta}_q}[C]) \cdot (NI - 1) \\ & + \max\Big(\max_{state}(tANS_{\hat{\theta}_q}[C]) \\ & + z\_bits,\ \max_{state}(tANS_{\hat{\theta}_q}[C-1])\Big) \\ \leq \; & ANS\_state\_bits \cdot NI + z\_bits \qquad (11) \end{aligned}$$

Here, $tANS_{\hat{\theta}_q}$ is the ANS table for $\hat{\theta}_q$ storing the number of bits to send to the output, which is addressed by the symbol and ANS state (omitted in the equation). This is an upper bound as, after an ANS symbol is coded, only a subset of the state domain is possible. To have the exact maximum code length for a given $\hat{\theta}_q$, the sub-symbol sequences used in eq. 11 can be coded, iterating over the state domain to set the initial state.

However, upper bounds on long sequences of symbols (like the coder block size) are more useful to size the output buffer. Then, if the ANS tables are already generated, using a simulation, the buffer could be computed such that there is no possibility of exceeding its size. A complete block of the symbol that produces the maximum code length should be coded for each $\hat{\theta}_q$, taking the largest binary block to size the output buffer. Compared to eq. 11, this procedure would produce tighter upper bounds.

If the ANS tables are not yet generated, the worst case could be assumed, in which all the max functions applied over $table_{ANS}$ in eq. 11 are equal to the ANS state number of bits. This would be accurate if the entropy of the C subsymbol, is close to the state bits. If that is not the case, entropy values plus safety margins (see section II-B) could be used instead of the $\max_{state}(tANS_{\hat{\theta}_q}[\,][\,])$ functions to do the estimation.

3) INTERACTIONS BETWEEN THE CODER AND THE REST OF THE SYSTEM

Something that should be noticed is that there is a negative feedback loop in place. Code expansion and long iterations are due to a large z given the estimated $\hat{\theta}_q$ for the context and $C = f(\hat{\theta}_q)$. If this situation persists, the large errors are going to drive $\hat{\theta}_q$ up, and subsequent large z that belong to this context would produce fewer bits, and fewer iterations as, in general, C can be increased with $\hat{\theta}_q$ for a given ANS state size. If C is not increased, as it might be limited for resource requirement reasons, the number of iterations would remain the same, but maximum code length will tend to decrease with increasing $\hat{\theta}_q$ estimations, as the number of bits used to code the sub-symbol C decreases.

For this reason, the actual largest binary output depends on the relative values of the block size, context domain size (number of context defined by surrounding quantized gradients), context $\hat{\theta}_q$ parameter estimation inertia, NI, C for each $\hat{\theta}_q$. The smaller the block size, the bigger context domain size, and the higher $\hat{\theta}_q$ parameter estimation inertia, the closer it gets to the limit established by eq. 11.

VI. DISTRIBUTION PARAMETERS ESTIMATION

To integrate the presented coder with JPEG-LS, the parameter estimation procedures to obtain $\hat{\theta}_q$ and $\hat{p}_q$ need to be introduced. These procedures not only estimate the distribution parameters, but also define the Q_p and Q_θ parameter quantization functions.

A. P PARAMETER ESTIMATION

An approximation of eq. 5 can be used to obtain $\hat{p}_q$. For this, the Nt sum is kept for each context and the bias cancellation procedure with some minor modifications can be employed, implementing a quantizer with uniform bin sizes. The reconstruction values can be chosen to minimize the KLD within each bin.

In Fig. 9, N is the context counter also used for the bias cancellation and p_{id} is $\hat{p}_q$ id number, which is also kept for each context. The parameters of this algorithm are Nt_p and the bound functions, $fi(N)$ and $fs(N)$. Nt_p determines the number of fractional bits stored in the N_t register and, as a consequence, the size of each quantization bin is 2^{-Nt_p} and $L_s - L_i = fs(N) - fi(N) = N$.

```
Require: $y, N_t, N, p_{id}$
 1: $N \leftarrow N + 1$
 2: $N_t \leftarrow N_t - p_{id} + (y << Nt_p)$
 3: $Li \leftarrow fi(N)$
 4: $Ls \leftarrow fs(N)$
 5: if $(Li > N_t)$ then
 6:     $p_{id} \leftarrow p_{id} - 1$
 7:     $N_t \leftarrow N_t + N$
 8: else if $(N_t >= Ls)$ then
 9:     $p_{id} \leftarrow p_{id} + 1$
10:     $N_t \leftarrow N_t - N$
11: end if
```

FIGURE 9. Update procedure the estimation of *p* Bernoulli parameter after accessing the context.

There are several ways (L_i, L_s) can be set. The extreme cases are analyzed, that is, the case $(-N/2, N/2)$ (where $p_{id}/2^{Nt_p}$ is centered within bin bounds) and $(0, N)$ (where $p_{id}/2^{Nt_p}$ is not centered, but equal to the lower bound). Fig. 10 shows the KLD for each of these cases for $p \in [0, 0.5]$ and $Nt_p = 4$. Here, the reconstruction values were chosen to be in the center of the bin, except for bin 0 of the centered case, where the reconstruction is computed taking $p = 0$ as the lower bound.

Although slightly more complex, choosing the $(-N/2, N/2)$ bounds allows a lower KLD for a given precision as it has a smaller bin (half the size) in the lower end of p range, where the KLD is more sensitive. This comes at the cost of an additional bin in the $p \in [0, 0.5]$ range (resulting in $2^{Nt_p - 1} + 1$ bins). However, the optimal reconstruction value of the upper bin (bin 2^{Nt_p-1}) is $p = 0.5$, so the tANS coder can be bypassed as y does not need to be coded (entropy = 1). Then, the number of tables required for both configurations is 2^{Nt_p-1}. It has to be noticed that low $\hat{p}$ ranges are used

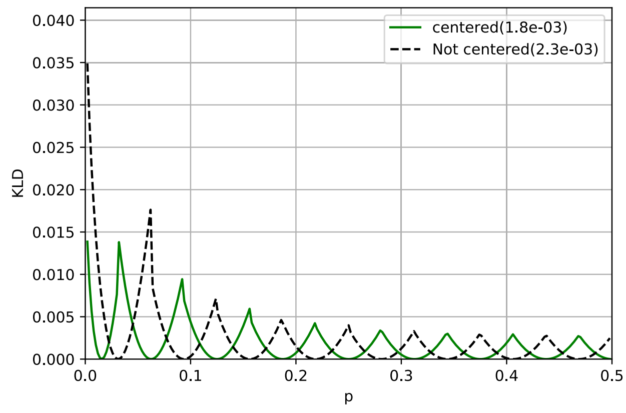

FIGURE 10. KL Divergence result of the quantization of the $\hat{p}$ parameter estimation for "centered" bin bounds $((id-1/2)/2^N t_p, (id+1/2)/2^N t_p)$ and "not centered" bounds $(id/2^{Nt_p}, (id+1)/2^{Nt_p})$ using $Nt_p = 4$ (accumulator precision). The average KLD, in bits, are shown between parentheses.

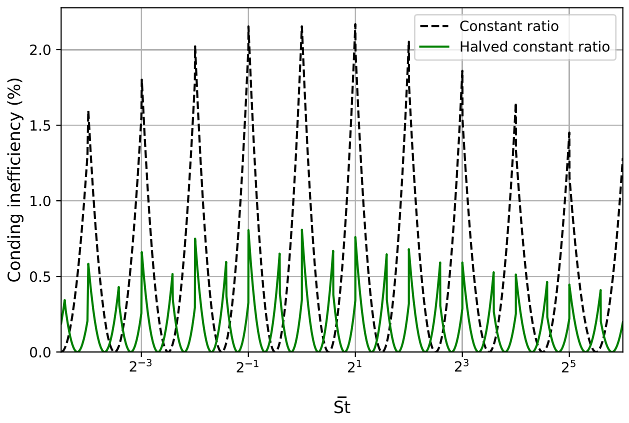

FIGURE 11. Coding inefficiency due to the quantization of $\overline{S_t}$ for two simple quantizers.

for low θ, so the impact of choosing one or the other would be appreciated for lower entropy cases (high *NEAR* and/or images that are accurately predicted, like smooth surfaces).

For photographic images, as forcing $\hat{p}_q <= 0.5$ does not increase the bpp, the condition $p_{id} < (2^{Nt_p-1} - 1)$ can be added to the `else if` (Fig. 9), avoiding the need to implement the logic to code y for the rare cases when $\hat{p}_q > 0.5$ (as indicated in section V-A).

B. θ PARAMETER ESTIMATION

Unlike the GPO2 coder used in LOCO-I, where the quantization of the TSG distribution parameters has to be adapted, particularly, to the k Rice parameter, with the proposed coder any quantization can be chosen. However, it is necessary to find a good trade-off between coding efficiency and coder resources.

An approximation of equation 6 is used to estimate θ. To implement it, a S_t sum register is stored for each context. Then, $\overline{St}$ needs to be computed and quantized, obtaining indexes which can be directly mapped to θ using eq. 6 with α and β set to 0. Given this direct relationship between $\overline{St}$ and θ, these two terms are used interchangeably.

1) CONSTANT RATIO QUANTIZER

The *constant ratio quantizer*, CRQ, is defined here as having the lower and upper bounds of each bin computed as $(Li, Ls) = (\overline{St_x} * r, \overline{St_x}/r)$, where $\overline{St_x}$ is the reconstruction value of the bin x and $r \in (0, 1)$ is a constant the regulates the size of the bins. This quantizer tends to keep the average KLD per bin constant when it is applied to $\overline{S_t}$. Once $\overline{St_0}$ and r are set, all bins bounds and reconstruction values can be determined. The bin bounds can be placed at $2^i, i \in \mathbb{Z}$, to obtain the quantization function used in LOCO-I for the average absolute error. However, as the presented coder is able to handle lower entropy distributions, the precision of the St register can be increased (as it was done for the N_t register) in order to support the quantization of $\overline{S_t} < 1$, which has more impact as *NEAR* increases.

The inefficiency ($KLD/Entropy$) due to the quantization of $\overline{S_t}$ can be observed in Fig. 11, where θ is computed as $\overline{S_t}/(\overline{S_t} + 1)$ and reconstruction values are computed using the rule stated above. Although those are not the optimal reconstruction values, they are close to them. Assuming a uniform distribution of $\overline{S_t}$ in the observed range, this simple quantization has on average inefficiency of 0.48%.

The division and quantization procedure can be carried out in several ways. LOCO-I presents an iterative method, implemented with a one-line for loop. Alternatively, the procedure in Fig. 9 can be adapted to accomplish the same quantization. To do this, L_i is set to 0, $Li = N \ll f(\theta_{id}, St_p)$ (where St_p is the precision of the S_t register) and in line 2, instead of subtracting θ_{id}, the bin lower bound needs to be computed based on θ_{id} and St_p.

These two procedures will not always output the same result, as the latter can only produce a decrement/increment of θ_{id} of 1 with respect to the previous id (this can be extended at the cost of more logic). In addition, this method requires storing θ_{id} in the context, although the size of the S_t register will be small as it will contain only the division residual.

In a software implementation, particularly a single thread one, this procedure will tend to be faster compared to the iterative one. However, from the hardware perspective, despite its iterative nature, the first alternative is appealing as it can be carried out outside the error quantization loop, and then the system throughput will tend to be higher. In this case, S_t and N are sent to the next stage where a possibly highly pipelined module obtains θ_{id}, while the context is being updated and a new sample is processed by the image quantizer. Whereas for the second alternative, the quantization procedure and update of the S_t index in the context needs to be completed in order to continue with next image sample.

2) FINER GRAIN AVERAGE QUANTIZERS

If higher coding efficiency is required, maintaining a simple quantization logic, the previously obtained quantization bins

```
Require: $S_t$
Require: $N$
Ensure: $\theta_{id}$
1: $S_t \leftarrow S_t + (z << St_p)$ {In the update phase}
2: $\theta_{id} \leftarrow 0$
3: $l \leftarrow N$
4: while $S_t > l$ do
5:     $\theta_{id} \leftarrow \theta_{id} + 2$
6:     $l \leftarrow l << 1$
7: end while
8: if $S_t > l - ((l + 2) >> 2)$ then
9:     $\theta_{id} \leftarrow \theta_{id} + 1$
10: end if
```

FIGURE 12. Procedure to obtain the quantized estimation parameter of the geometric distribution, $\hat{\theta}_q$, using the halved constant ratio quantizer.

can be uniformly divided. This can be implemented in several ways, for example, see Fig. 12. This algorithm halves each bin, generating the inefficiency curve labeled "Half constant ratio" in Fig. 11 when using optimal reconstruction values. It achieves an average inefficiency of 0.12% at the cost of doubling the number of required tables for a given $\overline{S_t}$ range.

C. RESETS

As done in JPEG-LS, the context count N and accumulators (in this case S_t and N_t) are halved when N reaches $N_0 = 2^i, i \in \mathbb{N}$ to limit the size of the registers and better adapt to changes in the context statistics.

VII. SELECTION OF CODER PARAMETERS

Different scenarios might need different trade-offs between resources, code efficiency, throughput and latency, and requiring support for a variable set of *NEAR* settings and types of images. However, it is not an easy task to establish the best configuration because of the strong coupling between the parameters of the coder.

These parameters are:

- The ANS state size, which sets limits to the ranges of possible $\hat{\theta}_q$ and $\hat{p}_q$ values, as well as, the maximum C for a given $\hat{\theta}_q$.
- The precision of the S_t accumulator, St_p, which sets a lower bound to the $\hat{\theta}_q$ values. If St_p is such that the lower bound it sets is equal or below the one set by the ANS state size, increasing St_p has almost not impact. The only effect it would have is that the accumulator will have an additional memory of past errors.
- The maximum $\hat{\theta}_q$ value.
- The Q_θ quantization function
- The precision of the N_t accumulator, Nt_p, which sets a lower bound on the $\hat{p}_q$ range. If only considering the centered uniform quantizer presented in section VI-A, Nt_p also determines the Q_p quantization function.
- The ANS table cardinality, C, for each $\hat{\theta}_q$.
- The code block size.
- The geometric coder maximum number of iterations, NI.

All system performance measurements are effected by all or most of the above parameters.

A. SELECTION METHODOLOGY

1) PRELIMINARY CONSIDERATIONS

A design methodology was derived from the mentioned relationships between coder parameters and the experience obtained when creating the prototype configurations for the experiments. For them, given an ANS state size, the main objective was to obtain configurations for a wide range of prediction error entropies. That is, to aim at a wide range of images and *NEAR* values. Additionally, for each ANS state size, a good trade-off between code efficiency and resources was sought. Then, the methodology intends to support the widest range of $\hat{\theta}$ and $\hat{p}$ for a given state size.

In addition, Nt_p is set so that it does not limit the range of $\hat{p}$, but not increasing it beyond that point as the impact on efficiency tends to be minimal while the number of tables doubles for each additional bit of precision (if the quantizer is configured to obtain the maximum number of quantization bins given the selected precision). Also, by default, the constant ratio quantizer is used for $\overline{S_t}$ and the centered uniform quantizer for p.

The maximum $\hat{\theta}$ that has practical implications to code efficiency is affected by the minimum *NEAR* supported, the pixel depth and the type of images to encode (classifying them according to their entropy, given the chosen model). Assuming that the actual z distribution as a geometric conditioned with the maximum possible value $(2^{ibits-1} - 1)$, then as θ tends to 1, then $\overline{S_t}$ will tend to $(2^{ibits-1} - 1)/2$ (half of the range). Fig. 6 shows that for the 8-bit gray images of the dataset some pixels reach this maximum (less than 0.4% of them). However, if only photographic images are considered, just 0.16% of the pixels reach a $\overline{S_t} > 16$. Then, to achieve high coding efficiency, the maximum $\hat{\theta}_q$ should correspond to a quantization bin that covers or is above $\overline{S_t} = 32$, in the general case, and $\overline{S_t} = 16$ for photographic images. If the minimum $NEAR > 0$, these values would be approximately scaled down by $\delta = (2 \cdot \min NEAR) + 1$.

To create the ANS tables a slightly modified version of the heuristic algorithm (mentioned in section II-B) was used. The goal of this modification is to ensure that the resulting table is a valid one, given the cardinality of the symbol source. This is done detecting if the original algorithm fails to assign at least 1 state to each symbol and then forcing it. In these cases, the KLD is expected to be higher than what eq. 1 states given that the tuning of the table to the set of symbols probabilities would tend to be worst. These tables are referred to as *suboptimal tables*. Note that this table generation algorithm can be improved.

2) METHODOLOGY

The methodology is as follows:

1) Choose the ANS state bits.
2) Set St_p and Nt_p such that they do not increase the lower bound on the range of the distribution parameter

TABLE 4. Prototype configurations used in the experiments.

Conf. name	State bits	St_p	Max $\overline{S_{tq}}(\hat{\theta}_q)$	Q_θ	Nt_p	C range	# of θ tables	# of p tables	total # of rows
Nt4_Stcg5_ANS4	4	5	22.63 (0.958)	CG	4	1-8	11	8	304
Nt5_Stcg6_ANS5	5	6	90.51 (0.989)	CG	5	1-8	14	16	960
Nt5_Stfg6_ANS5	5	6	55.62 (0.982)	FG	5	1-8	26	16	1344
Nt6_Stcg7_ANS6	6	7	90.51 (0.989)	CG	6	1-8	15	32	3008
Nt6_Stcg8_ANS7	7	8	90.51 (0.989)	CG	6	1-8	16	32	6144
Nt6_Stfg8_ANS7	7	8	111.24 (0.991)	FG	6	1-8	32	32	8192

estimations they affect ($\hat{\theta}$ and $\hat{p}$, resp.), given the selected ANS state size. For this, start with small precision, for example, set both St_p and Nt_p to 0. Then, using $C = 1$, try to generate the ANS tables for the first bin of each quantizer (smallest $\hat{\theta}_q$ and $\hat{p}_q$). If it succeeds, increment the corresponding accumulator precision. If it returns a suboptimal table, stop.

3) Set the maximum $\hat{\theta}_q$. Choose the minimum $\hat{\theta}_q$ between the maximum one that has practical implications to compression and the maximum supported by the ANS state size. To check this latter maximum, proceed similarly to step 2, iterating over the order set of $\hat{\theta}_q$ until a suboptimal table is returned. Particularly for hardware implementations where ANS tables would be stored in on-chip memory, using a number of $\overline{S_t}$ quantization bins different to a power of 2 will result in unused resources. Then, if the ANS state size allows it, increasing the number of $\overline{S_t}$ quantization bins up to a power of 2 might provide some additional compression without requiring more resources.
4) For each $\hat{\theta}_q$, choose the maximum allowed C. For this, proceed similarly to step 2, but in this case, start with $C = 1$ and iterate over power of 2. In the experiments, the maximum allowed C (8) was not big enough to have a significant intrinsic ANS KLD (see eq. 1). However, depending on the implementation, the maximum used C can significantly affect memory resources and, particularly in the case of hardware implementations, coder throughput. For this reason, an upper limit to C may be set using the ANS state bits and the number of required ANS tables to do resource and performance estimations.

Initially, to choose the ANS state bits, it can be assessed the number of bits that can be afforded given the memory resources and performance requirements. This should be done assuming that 16-64 ANS tables would be employed. For FPGA implementations, the results in [30] can be used as a guide to understand the impact of resources on performance.

3) SETTING THE CODE BLOCK SIZE AND *NI*

The code block size is relatively decoupled from the rest of parameters. The larger it is set, the better compression ratio achieved. However, if the binary is aligned to bytes and the ANS state bits is below 10, no significant improvement will result increasing the code block size above the tens of thousands of symbols. Increasing its size comes at the cost of more memory resources and some impact on latency, although, in practical scenarios, this does not represent a major problem to achieve high efficiency and low latency. For most of the experiments, presented in section VIII, the block size was set to 2048 as it results, on average, in a KLD of 0.005 bits or less when binary blocks are aligned to bytes (ANS state bits $\in [4..7]$). Moreover, for an FPGA implementation processing 8-bit images and using 32 $\hat{p}_q$ tables and 32 $\hat{\theta}_q$ tables, one block of symbols with their distribution parameter estimations can be stored in 1 Xilinx 36K BRAM or 2 Intel M20K.

Regarding *NI*, a larger value tends to reduce the bpp. However, as in the case of the code block size, increasing *NI* has diminishing returns. Also, the worst-case throughput worsens linearly (initially) with *NI*. Most test were run with *NI* set to 7, the number of bits required to represent z variable for 8-bit images. Then, the worst case throughput is the same as the resulting from a coder using trivial binarization. Using this value, little negative impact on compression is seen in general, and almost no impact for photographic images.

B. TUNING THE CODER PARAMETERS

If the application needs to support a limited number of *NEAR* values and/or type of images, then better trade-offs between code efficiency and resources could be obtained.

Using the halved constant ratio quantizer for $\overline{S_t}$ (doubling the number of bins covering the same range) can have a greater impact on code efficiency than increasing the ANS state bits, while in former equal or less memory resources are required. This effect can be observed in the experimental results shown in the following section.

Similarly, for $\hat{p}$, changing the centered uniform quantizer with the non-centered one and incrementing Nt_p in one, would result in the same minimum $\hat{p}_q$, but the rest of the $\hat{p}$ range would have bins that are half the size, resulting in a smaller KLD.

If performance models and resource restrictions are available, the optimization task could be handled algorithmically.

VIII. EXPERIMENTAL RESULTS

A prototype of LOCO-ANS was implemented using C++, which was tested with a set of different configurations with the goal of exploring the design space. The tested configurations can be seen in Table 4. The name of the configurations indicates the most relevant parameter settings, following the format:

TABLE 5. Mean bpp and iterations obtained using the prototype configurations with NI = 7.

	Nt4_Stcg5_ANS4		Nt5_Stcg6_ANS5		Nt5_Stfg6_ANS5		Nt6_Stcg7_ANS6		Nt6_Stcg8_ANS7		Nt6_Stfg8_ANS7	
Error	bpp [1]	$\bar{i}$	bpp [1]	$\bar{i}$	bpp [1]	$\bar{i}$	bpp [1]	$\bar{i}$	bpp [1]	$\bar{i}$	bpp [1]	$\bar{i}$
0	3.41 (-2.8%)	1.3	3.31 (0.4%)	1.3	3.3 (0.7%)	1.3	3.29 (0.9%)	1.3	3.29 (0.9%)	1.3	3.28 (1.2%)	1.3
1	2.04 (3.5%)	1.2	2.03 (4.2%)	1.2	2.02 (4.5%)	1.1	2.02 (4.4%)	1.1	2.02 (4.5%)	1.1	2.01 (4.8%)	1.1
2	1.55 (5.9%)	1.1	1.53 (7%)	1.1	1.53 (7.2%)	1.1	1.53 (7.3%)	1.1	1.53 (7.4%)	1.1	1.52 (7.7%)	1.1
3	1.28 (8.4%)	1.1	1.26 (9.9%)	1.1	1.26 (10.1%)	1.1	1.25 (10.4%)	1.1	1.25 (10.4%)	1.1	1.25 (10.7%)	1.0
4	1.1 (10.9%)	1.1	1.07 (12.8%)	1.1	1.07 (13%)	1.1	1.07 (13.5%)	1.0	1.06 (13.5%)	1.0	1.06 (13.8%)	1.0
5	0.96 (13.2%)	1.1	0.94 (15.4%)	1.1	0.94 (15.6%)	1.1	0.93 (16.3%)	1.0	0.93 (16.4%)	1.0	0.92 (16.6%)	1.0
6	0.86 (14.8%)	1.1	0.83 (17.4%)	1.1	0.83 (17.6%)	1.1	0.82 (18.4%)	1.0	0.82 (18.6%)	1.0	0.82 (18.9%)	1.0
7	0.77 (15.9%)	1.0	0.75 (18.9%)	1.0	0.74 (19.1%)	1.0	0.73 (20.1%)	1.0	0.73 (20.3%)	1.0	0.73 (20.5%)	1.0
8	0.7 (16.6%)	1.0	0.67 (19.7%)	1.0	0.67 (20%)	1.0	0.66 (21.1%)	1.0	0.66 (21.4%)	1.0	0.66 (21.7%)	1.0
9	0.65 (17.5%)	1.0	0.62 (21.3%)	1.0	0.62 (21.6%)	1.0	0.61 (22.9%)	1.0	0.6 (23.2%)	1.0	0.6 (23.4%)	1.0
10	0.6 (18.2%)	1.0	0.57 (22.7%)	1.0	0.56 (23.1%)	1.0	0.55 (24.5%)	1.0	0.55 (24.8%)	1.0	0.55 (25%)	1.0

The results presented here were obtained averaging the bpp computed for each image, so that all images have the same weight in the average.
Code block size set to 2048 symbols.
[1] The improvement percentage compared to JPEG-LS is shown between parentheses.

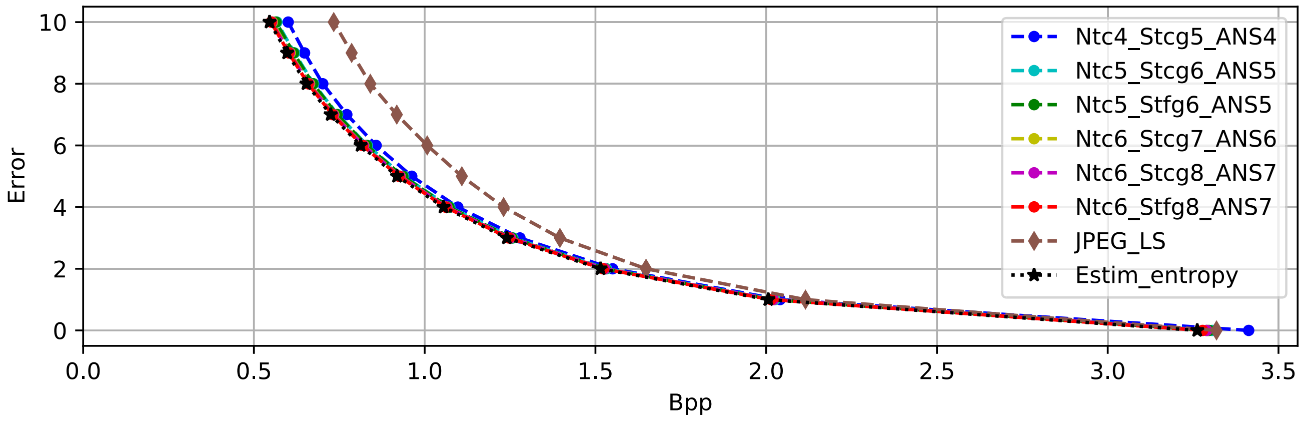

FIGURE 13. Mean bpp obtained using the prototype configurations with NI = 7 vs. JPEG-LS.

Nt$\{Nt_p\}$_St$\{Q_\theta\}\{St_p\}$_ANS{State bits}, where Q_θ is "cg" for the (coarse grain) constant ratio quantizer or "fg" for the (finer grain) halved constant ratio quantizer described in section VI-B. In all cases, N_0 was set to 64 (JPEG-LS default), the centered uniform quantizer was used to quantize the distribution parameter estimation $\hat{p}$ and the maximum C was set to 8.

Apart from the configuration parameters, Table 4 shows the total number of rows (total number of tables by the number of states), which provides a measure of the memory resources required by each configuration (the actual memory utilization depends on the implementation).

A. ANALYSIS OF LOCO-ANS CONFIGURATIONS PERFORMANCE

The average compression results (over the whole dataset) can be seen in Table 5 and plotted in Fig. 13. Additionally, the entropy estimation (according to the model) is shown in the figure to appreciate the efficiency of the configurations.

For these experiments, the code block size was set to 2048 and *NI* to 7. All configurations surpass JPEG-LS mean compression ratios for all the tested *NEAR* settings, except for the Nt4_Stcg5_ANS4 configuration for lossless. The highest performing configuration, Nt6_Stfg8_ANS7, achieves a 1.2% mean bpp improvement for lossless, which increases with *NEAR*. Interestingly, even the lighter version, Nt4_Stcg5_ANS4, is able to obtain remarkable reductions of bpp for near-lossless compression, with improvements ranging from 3.5% for $NEAR = 1$ to 18.2% for $NEAR = 10$. However, for $NEAR > 10$ the improvement percentage for this particular configuration starts to decrease, as the lower entropy distributions require larger ANS state sizes and higher precision estimations.

1) COMPRESSION OF PHOTOGRAPHIC IMAGES

It is worth noting that when only considering the photographic images of the dataset, the bpp improvements are greater. In this case, as observed in Table 6, even the configuration using 4 bits for the ANS state size outperforms JPEG-LS for all the tested *NEAR* values, including lossless.

2) EFFECT OF ITERATIONS LIMITATION

The results in Table 5 and 6 correspond to configurations with $NI = 7$. When the number of iterations of the geometric coder are not limited, the compression ratio slightly increases for lossless compression of the complete dataset, allowing

TABLE 6. Mean bpp for photographic images of the dataset obtained using a selection of the prototype configurations with NI = 7.

Error	JPEG_LS	Nt4_Stcg5_ANS4	Nt6_Stfg8_ANS7
0	3.20	3.18 (0.5%)	3.15 (1.4%)
1	1.95	1.86 (4.9%)	1.84 (5.8%)
2	1.48	1.36 (8.0%)	1.33 (9.7%)
3	1.22	1.08 (11.5%)	1.05 (13.9%)
4	1.06	0.89 (15.5%)	0.86 (18.6%)
5	0.94	0.76 (18.9%)	0.73 (22.7%)
6	0.84	0.66 (21.6%)	0.62 (26.1%)
7	0.76	0.58 (23.9%)	0.54 (29.1%)
8	0.69	0.51 (25.6%)	0.47 (31.7%)
9	0.63	0.46 (26.9%)	0.41 (34.1%)
10	0.58	0.41 (28.2%)	0.36 (36.8%)

The improvement percentage compared to JPEG-LS is shown between parentheses

the Nt6_Stfg8_ANS7 configuration to reach a 1.3% improvement over JPEG-LS. However, for near-lossless compression or for the photographic images (including lossless), there is not a practical difference in compression when setting $NI = 7$ compared to not limiting the iterations.

3) ANALYSIS AT THE IMAGE LEVEL

A comparison at the image level is presented in Table 7, which shows the number of images of the dataset (and of the photographic image subset in parentheses) JPEG-LS obtains better compression ratios for different error tolerances. The numbers observed for $NEAR = 5$ repeat exactly up to $NEAR = 12$. From that point, configurations with smaller ANS states start to struggle with lower entropy images, which can also be appreciated in Fig. 13.

The synthetic image "zone plate" is the hardest to compress (according to the model) and the one where JPEG-LS tends to outperform LOCO-ANS. As mentioned in section III, because of the change in the gradient quantization function, the entropy estimation for all $NEAR > 0$ worsens for this particular synthetic image. This results in JPEG-LS obtaining a bpp below the estimated entropy, according to the modified model for most $NEAR > 0$. For the best performing LOCO-ANS configuration in Table 7, this situation occurs for all the cases in which JPEG-LS obtained a better compression ratio, except for one case where the average estimated entropy is 0.0001 bits lower that JPEG-LS bpp. Then, in these cases, the problem lies in the statistical model (which is better suited for photographic images) and not in the coder.

The best performing configuration introduced in Table 7 has an increased block size of 16384 symbols. This reduces the KLD due to the need of sending the final ANS state at the end of a code block and aligning each new block to a word, in this case, to bytes (eq. 8). For this reason, the configuration achieves a 1.5%, 5% and 25.7% mean bpp improvement for $NEAR$ set to 0, 1 and 10, respectively, when compressing the complete dataset. These improvements increase to 1.6%, 6% and 37.6% when only taking into account photographic images.

TABLE 7. Number of images of the dataset that JPEG-LS achieves a lower bpp. Dataset size: 14 images.

Error	0 [1]	1	2	3	4	5
Nt4_Stcg5_ANS4	8(6)	3(1)	3(1)	3(1)	3(1)	3(1)
Nt4_Stcg5_ANS4 [2]	8(6)	3(1)	3(1)	3(1)	3(1)	3(1)
Nt5_Stcg6_ANS5	4(3)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt5_Stcg6_ANS5 [2]	5(3)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt5_Stfg6_ANS5	1(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt5_Stfg6_ANS5 [2]	2(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stcg7_ANS6	1(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stcg7_ANS6 [2]	0(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stcg8_ANS7	1(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stcg8_ANS7 [2]	0(0)	2(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stfg8_ANS7	1(0)	1(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stfg8_ANS7 [2]	0(0)	1(1)	1(0)	2(1)	1(0)	1(0)
Nt6_Stfg8_ANS7 [2 3]	0(0)	1(1)	1(0)	1(0)	1(0)	1(0)

By default, NI is set to 7 and block size to 2048.
[1] The number of images, considering only photographic ones, is shown between parentheses (total of photographic images is 12).
[2] Configuration with unlimited iterations.
[3] The configuration has a block size of 16384.

B. EXPERIMENTAL SYSTEM EFFICIENCY

To evaluate experimentally the sources of inefficiencies, given the chosen model, the KLD resulting from parameter estimation procedures and from the coder were decoupled. To do this, for each image sample, a second average entropy computation was performed, denoted as $H(TSG(\hat{\theta}_q, \hat{p}_q))$, which estimates the bpps assuming an ideal coder. This entropy was computed using the quantized estimations of the distribution parameters (obtained with the procedures described in section VI), instead of using the optimal estimators $\hat{\theta}$ and $\hat{p}$ (computed using eq. 5 and 6). Then, the KLD due to the distribution parameters estimation procedures was computed as $H(TSG(\hat{\theta}_q), \hat{p}_q)) - H(TSG(\hat{\theta}, \hat{p}))$ and the KLD due to the coder as $bpp - H(TSG(\hat{\theta}_q), \hat{p}_q))$.

The resulting KLD, for all images and for $NEAR \in [0..20]$ is shown in Figs. 14 and 15. These were plotted as a function of $H(TSG(\hat{\theta}, \hat{p}))$. A logarithmic scale is used for the KLD axis, given that values in this axis range over 5 orders of magnitude. The entropies and bpp resulting from the experiments were stored using 4 fractional digits, as it was considered that increasing it would not provide any useful information. Because of this, 10^{-4} is the smallest difference that can be appreciated in the log scale, smaller values (zero or negative) are plotted with a KLD $= 6 \cdot 10^{-5}$. In addition, this explains the patterns in the $10^{-4} \leq KLD \leq 10^{-3}$ range.

1) PARAMETER ESTIMATION EFFICIENCY

In Fig. 14, in general, it can be observed that high efficiencies are achieved by the distribution parameter estimation procedures. For an image average entropy greater than 1, the quantization of $\hat{\theta}$ (indirectly as $\overline{S_t}$ is quantizated) dominates observed inefficiency. Here, a clear separation between prototypes using the coarse grain and the fine grain quantizer for $\overline{S_t}$ can be seen. As a consequence of this effect,

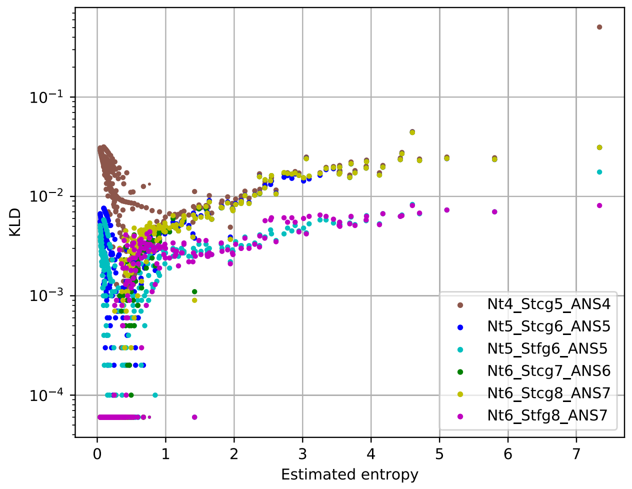

FIGURE 14. KL Divergence as a function of the estimated entropy due to distribution parameter estimation inefficiencies for all images in the dataset for *NEAR* ∈ [0..20]. The divergence results from using the quantized estimations of the distribution parameters (computed as described in section VI) instead of the estimations obtained using eq. 5 and 6.

Nt5_Stfg6_ANS5 configuration is capable of matching, and in some cases improving, the compression ratios achieved by the Nt6_Stcg7_ANS6 configuration, whereas the latter requires about twice the memory resources. The finer $\overline{S_t}$ quantization allows the former to be more efficient for medium entropies.

However, when the average entropy diminishes below 1, the effect of having a minimum $\hat{\theta}_q$ and a minimum $\hat{p}_q$ starts to be noticeable (the entropy diminishes as $\theta \to 0$ and p moves away from .5). Here, the main parameter that separates the points of the plot is the ANS state size, which determines these minimums. Additionally, the quantization of $\hat{p}$ contributes to the increase of the KLD as it is less efficient in this zone (observed in Fig. 10).

On the other end of the range, for high entropies, the effect of having a maximum $\hat{\theta}_q$ would also increase the KLD. This can only be observed for the prototype using a 4-bit ANS state when losslessly coding the greatest entropy image (zone plate).

2) CODER EFFICIENCY

In the case of the coder KLD, seen in Fig. 15, the relevant parameters are the ANS state size, the code block size and NI. For an entropy in the $(1, 5)$ range, the KLD is basically flat, with a small positive slope, and it would mainly come from ANS intrinsic KLD (eq. 1) and the code block size (eq. 8). The magnitude of the KLD due to the latter can be appreciated comparing the Nt6_Stfg8_ANS7 prototype with block sizes of 2048 and 16384 symbols (both with unlimited iterations). Note, however, that these differences are not as big as the plot might suggest, given that the KLD is plotted on a logarithmic scale. Additionally, notice that with a block size of 16384 symbols the coder of this prototype achieves a practically null KLD.

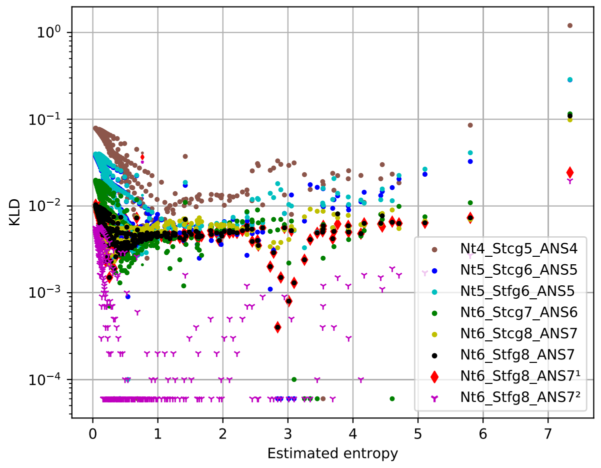

FIGURE 15. KL Divergence as a function of the estimated entropy due to coder inefficiencies for all images in the dataset for *NEAR* ∈ [0..20]. Computed as the average bits produced by the coder minus the average entropy assuming the quantized estimations of the distribution parameters are optimal. Default configuration: NI = 7 and code block size = 2048 symbols. [1]configuration with unlimited iterations [2]configuration with unlimited iterations and code block size = 16384 symbols.

The increase in the KLD observed for the lower entropies is due to the use of the suboptimal tables for the smaller $\hat{\theta}_q$ and $\hat{p}_q$ (see the selection methodology in section VII). The lower the average entropy, the more probable is to use these tables, then the KLD increases. Although the use of these suboptimal tables increases the coder KLD, including these smaller distribution parameters more than compensates, then, the final effect is a reduction in the overall KLD.

As the entropy increases, the KLD can increase for several reasons: the limitation of the geometrical coder iterations (eq. 10), the increase of C as $\hat{\theta}_q$ increases (which in turn increases the intrinsic ANS KLD as indicated by eq. 1) and the increased average iterations (which make the coder incur in the ANS intrinsic KLD several times). The effect of NI can be observed comparing the points corresponding to the Nt6_Stfg8_ANS7 prototype with $NI = 7$ and with unlimited iterations. For low and medium entropies, the two configurations result in approximately the same KLD, while for higher ones the KLD due to the escape mechanism can be appreciated. To understand the magnitude of the effect of increasing C as $\hat{\theta}_q$ increases, the highest entropy cases were compressed with a modified Nt6_Stfg8_ANS7 prototype, setting $\max(C) = 4$. The resulting KLD (not shown in the figure) went back to the (0.001- 0.008) range when the iterations were not limited. For this modified configuration, the average iterations increased (nearly doubled).

C. SOFTWARE PERFORMANCE COMPARISON

In this section, we compare LOCO-ANS in terms of compression ratio and encoder/decoder speed against well-known and recently developed lossless and near-lossless codecs:

- JPEG-LS (implementation: [37]).
- CALIC (implementation: [38]).

- JPEG2000 Part 1 [39] (implementation: [40]) and JPEG2000 Part 15 High-throughput JPEG2000 (HTJ2K) [41] (implementation: [42]) (none of them provide near-lossless compression).
- WebP [43] (implementation: [44]).
- WebP2, currently under development (implementation: [45]).
- JPEG-XL [46], [47] (although not yet a standard, it is currently under evaluation and the reference software is available [48]).

The tests were carried out in a Raspberry Pi 3 Model B with 1 GB of RAM. This platform was chosen because it better resembles, compared to an x86_64 system, the memory and compute limitations that embedded systems tend to face, which is our target. In addition, it is a widespread platform, facilitating the reproducibility of the results here presented.

All tests were run using a single thread, given that most codec implementations do not have multi-threading capabilities, although they could support it. For example, images could be divided in tiles, like JPEG-XL or JPEG2000, which are able to do. In the case of JPEG-LS, although this is not part of the standard, this could be easily implemented, like in the hardware implementation presented in [9]. This tiling, when performed dividing the image vertically, not only allows for a higher level of parallelism but also tends to benefit JPEG-LS statistical modeling, thus increasing compression (demonstrated by the tests).

Additionally, to show other possible speed-compression trade-offs, a version of LOCO-ANS using four gradients to define the context, as in the original LOCO-I and the standard extension, was also included. Finally, the configurations used for the codecs can be seen in Table 8.

TABLE 8. Codec configurations used in the tests.

Codec	configurations
LOCO-ANS (conf. 1)	Nt6_Stcg8_ANS7, 3 grad [1]
LOCO-ANS (conf. 2)	Nt6_Stfg8_ANS7, 4 grad [1]
CALIC	Arithmetic coder
JPEG2000	lossless
HTJ2K	lossless
WebP	effort level(-z) =[1,2] [2]
WebP2	effort level(-effort)=[1] [2]
JPEG-XL	num_threads=0, modular, speed(-s)=[2..5] [2]

[1] NI was set to 7, tables for $p \in [0, 0.5]$ and a block size of 16384
[2] Other effort modes were supported, however they do not compare favorably against the presented configurations of the same codec or encoder times were well over an order of magnitude slower than LOCO-ANS

1) DATASET

Given the large memory requirements of JPEG-XL (even for the lower effort setting "-s0"), it was not possible to process the largest images of the Rawzor dataset using this codec in the chosen platform. For this reason and to obtain more robust results, these tests were run using gray versions of a subset of the Challenge on Learned Image Compression (CLIC) [49] training dataset (this subset was used in the evaluation of JPEG-XL lossless compression, and it is available in [50]). It contains 303 2048×1320 photographic images and was not used during the development of LOCO-ANS, so it is also good for validation purposes.

2) ANALYSIS

The results for encoder and decoder procedures are presented in Figs. 16 and 17, respectively. In addition, Table 9 summarizes the results for lossless compression, where entries are sorted by bpp. As expected for software implementations, the increased compression obtained by LOCO-ANS comes at the cost of a reduction in the encoder and decoder speeds compared to JPEG-LS. Specifically, the tests show a 32% and 46% encoder speed reduction and a 42% and 54% decoder speed reduction for lossless compression. As the peak error increases, both implementations run-times tend to decrease, although the relative comparison favors JPEG-LS, which can be explained by the incremented use of the run-length coder.

TABLE 9. Encoder/Decoder speed comparison for lossless compression.

Codec	bpp	Enc. BW (MiP/s)	Dec. BW (MiP/s)
JPEG-XL (s5)	3.62	0.20 [1]	1.58 [2]
JPEG-XL (s4)	3.62	0.25 [1]	1.58 [2]
CALIC	3.63	1.65 [1]	1.58
LOCO-ANS (2)	3.65	4.96 [1]	4.81 [2]
JPEG-XL (s3)	3.68	1.87	2.10
LOCO-ANS (1)	3.69	6.26 [1]	5.94 [2]
JPEG-LS	3.73	9.21 [1]	10.66 [2]
WebP2 (effort 1)	3.79	0.11	1.44
WebP (z2)	3.81	0.46	13.48 [2]
JPEG2000	3.82	1.76	2.13
WebP (z1)	3.88	0.65	13.59 [2]
JPEG-XL (s2)	3.98	3.22	4.00
HTJ2K	4.08	8.40	14.52 [2]

[1] No other codec simultaneously encodes faster and achieves a lower bpp (in Pareto frontier).
[2] No other codec simultaneously decodes faster and achieves a lower bpp (in Pareto frontier).

Despite this decrease in performance, given the codecs utilized in this comparison, both of the LOCO-ANS configurations presented are on the Pareto frontier [51] of encoder speed versus bpp and decoder speed versus bpp. When it comes to near-lossless compression, most codecs do not perform so well. Particularly, in the case of JPEG-XL, the near-lossless quantization is done as a preprocessing step that reduces the cardinality of the prediction errors, not the range of these errors, and then, it is up to the entropy encoder to detect and exploit the reduced error set cardinality. For this reason, the faster compression modes not only fail to increase compression, but they decrease it. Conversely, LOCO-ANS excels in this type of compression achieving the highest compression ratios for a given peak error, in the presented order of magnitude of encoder speed, and it is only surpassed in encoder speed by JPEG-LS and in decoder speed by JPEG-LS and WebP.

Lastly, to show that tilling does not worsen LOCO-ANS performance and given that the prototype supports it, tests

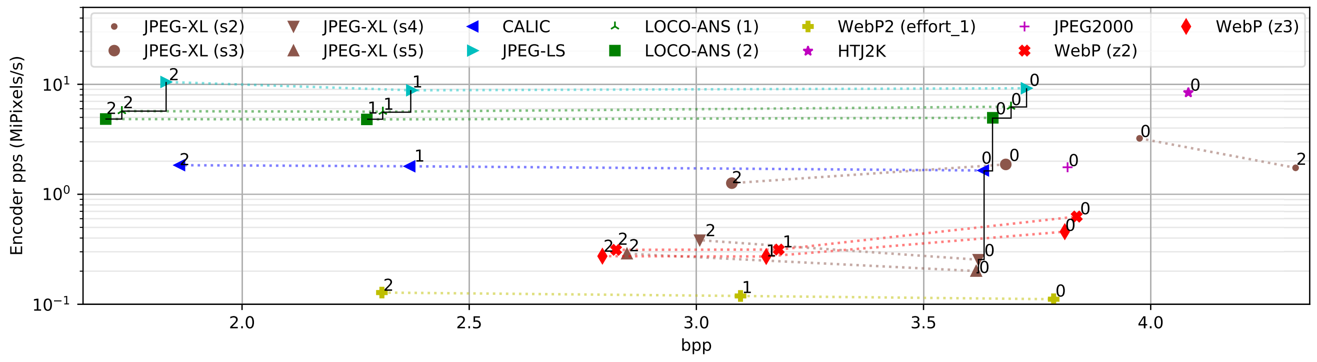

FIGURE 16. Average encoder MiPixels/s versus average bpp for software implementations of different codecs. Numbers next to each point indicate the corresponding peak error. Pareto frontier is drawn with a solid line for error tolerances ∈ {0, 1, 2}.

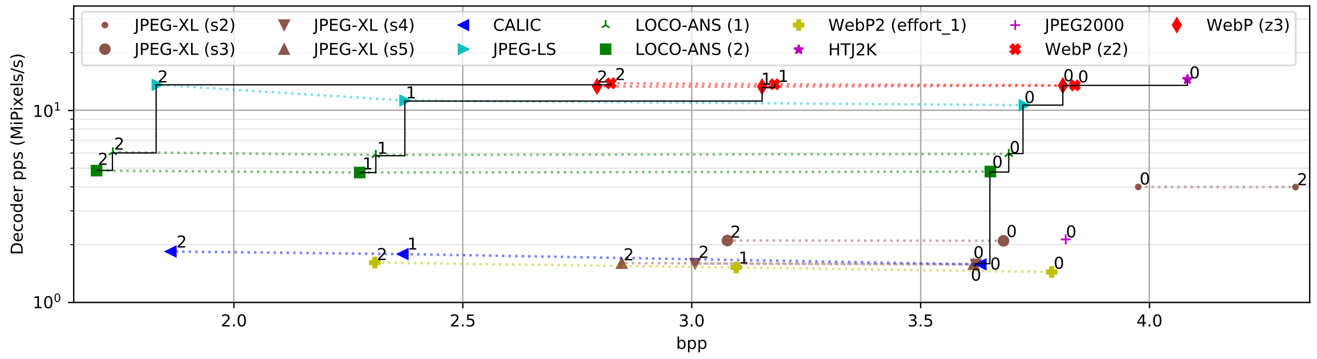

FIGURE 17. Average decoder MiPixels/s versus average bpp for software implementations of different codecs. Numbers next to each point indicate the corresponding peak error. Pareto frontier is drawn with a solid line for error tolerances ∈ {0, 1, 2}.

were also run dividing the images in 4 columns (number of cores available in the Raspberry Pi platform). As a result, 3.67 and 3.64 bpp (2.3% improvement compared to JPEG-LS) were obtained for configurations 1 and 2, respectively, compared to 3.69 and 3.65 bpp without tilling.

3) COMPARISON WITH OTHER ANS-BASED APPROACHES

Although JPEG-XL entropy encoder is also based on ANS and the implementation used in the tests is highly optimized, it runs slower (in general several times) than LOCO-ANS, even using low-effort modes. Moreover, CALIC compares favorably against it.

JPEG-XL uses a modified version of Range ANS (rANS) to encode symbols given clustered histograms. To perform this operation, all prediction residuals are computed (that is, for the complete image) and then histograms for each context are generated. In general, these contexts are dynamically determined (obtained at run-time). After that, histograms can be clustered (context merging) and the final histograms are signaled to the decoder. LUTs (Look-up Tables) to speed up rANS (with a not trivial initialization) are generated for each of these histograms (these LUTS are called *Alias Tables*, not to be confused with tANS tables). Finally, residuals (or more generally, tokens), after going through other numerical manipulations, are coded with rANS using a 32bit state. ANS code blocks coincide with a tile (256 × 256 pixels).

To our understanding, JPEG-XL aims to be a general-purpose codec, although oriented to web image delivery [52]. In this scenario, it is reasonable to allow higher complexity (higher computation time and/or computation/memory resources), particularly on the encoder side. This encoder vs decoder speed trade-off is also observed in WebP and WebP2. However, the aim of our work was to improve image compression in situations with stronger constraints (low resources, low energy budget, high throughput). It is easy to observe that, given their resource requirements, many of the sub-processes that JPEG-XL performs to code the generated tokens are not well suited for high-performance hardware nor embedded software implementation. This is also the case for other procedures that are part of the JPEG-XL codec, for example, those that require full image scanning.

In contrast, LOCO-ANS approach, based on static tANS using parametric distributions (instead of rANS using clustered histograms) leads to higher throughput (illustrated by Figs. 16 and 17). Additionally, it is more suitable for a hardware implementation, given that buffering is limited, simple arithmetic is used and tables are generated at compile time, which allows both software and hardware optimizations, particularly in the latter case.

D. EXPECTED HARDWARE IMPLEMENTATION THROUGHPUT

The software performance results do not translate directly to hardware implementations, given that encoders are normally implemented as pipelines, where pixel decorrelation and coding run in parallel, providing lower latency and higher speed in the compression. These features are very suitable for real-time applications [53].

The coder symbol rate is going to be directly affected by the mean number of iterations required to encode z, but, as it can be seen in Table 5, these tend to be very small for all configurations. Even considering lossless compression, including artificial images and using the worst performing configuration, 1.34 iterations/pixel is achieved. If a single ANS state is shared between both y and z coders (as in the experiments), the coder symbol rate is dependent on the mean number of accesses to the ANS tables per image sample, which adds one access to code y. Alternatively, two independent ANS coders could be used for y and z variables, decoupling their ANS state. This would allow to code y and z in parallel and to tune the state size for each of them separately. However, both final states should be sent at the end of the block, which can result in an increased KLD. If enough memory resources are available, this can be mitigated using a larger code block size.

As studied in [30], the memory resources required by the tables impact on the system throughput, as they correlate with the maximum operating frequency. Given the throughput obtained by these hardware implementations of ANS coders and the average cycles required by the coder, it is expected that the throughput of an FPGA implementation of the proposed TSG coder will outperform the best reported JPEG-LS implementation supporting near-lossless [9]. The throughput obtained by the JPEG-LS and the ANS encoder implementations are considered comparable as they targeted the same technology (Xilinx Virtex-6). Then, the utilization of this coder will increase compression, while the throughput bottleneck would remain in the context update and the pixel quantization (and reconstruction) procedures.

E. DISCUSSION

Given the obtained results, it is observed that the proposed TSG ANS coder is particularly well suited for sources with an entropy in the (.15, 4) range, approximately. Even, the 4-bit ANS state configuration achieves a great efficiency with low memory resources and capable of high-throughput operation. Taking into account the strengths of the GPO2 and the run-length coders, it would be interesting to combine these with the proposed coder. The resulting system may achieve the best complexity-efficiency trade-off for a very wide range of applications.

Additionally, it is worth noting that the TSG coder (or just the geometrical coder) could also be used in other applications, such as audio compression. For example, in the case of MPEG-4 ALS [54] or FLAC [55], the prediction error distribution could be modeled as a two-sided geometric.

IX. CONCLUSION

In this work, improved lossless and near-lossless compression was achieved through a series of modifications of the JPEG-LS standard. Particularly, the development of an ANS based coder for two-sided geometric sources provides highly efficient and low complexity coding. Additionally, this coder enabled the introduction of more precise distribution parameter procedures and to quantize more effectively the gradient defined context space.

The system as a whole admits a wide range of configurations, providing the capability to obtain different trade-offs between coding efficiency, resources and throughput, which allows it to be used in a variety of applications. A prototype available to the community was implemented and a set of experiments were run with different configurations to explore the design space. These configurations range from a very low resource instance that outperforms JPEG-LS in near-lossless compression to an instance using 64 tables with a 7-bit ANS state that closely approaches the estimated entropy.

When compared to JPEG-LS baseline compressing photographic images, LOCO-ANS, using the same context size, is able to achieve up to a $1.6\%, 6\%$ and 37.6% mean bpp improvement for an error tolerance set to 0, 1 and 10, respectively. Allowing an increase of the context size and image tiling, a 2.3% lower bpp is obtained for lossless compression. Moreover, LOCO-ANS approaches lossless compression rates of more complex encoders, even surpassing them in near-lossless compression, and obtaining a much faster encoder speed.

Given that many applications would benefit from a hardware implementation, future work will focus on developing and evaluating a prototype implemented in an FPGA, with higher throughput and lower latency, which can be used for real-time purposes.

APPENDIX I. NOTATION TABLE

TABLE 10. Notation used in this work.

Notation	Description
$NEAR$	Codec input that determines maximum absolute difference between the original pixel and the decoded one.
ϵ	Two-sided geometrically distributed variable
z	Geometrically distributed variable
y	Bernoulli distributed variable
s	Two-sided geometrically fractional bias
θ	Geometric and Two-sided geometrically shape parameter
p	Bernoulli distribution parameter
$\hat{\theta}_q$	Quantized estimation of θ
$\hat{p}_q$	Quantized estimation of p
Q_θ	θ quantization function
Q_p	p quantization function
C	Symbol alphabet cardinality of an ANS table, given a $\hat{\theta}_q$
S_t	Accumulator storing $\sum_{i=1}^{t} z_i$
$\overline{S_t}$	Mean of the z geometric variable $(\sum_{i=1}^{t} z_i/t)$
N_t	Accumulator storing $\sum_{i=1}^{t} y_i$
St_p	Number of fractional bits used by S_t
Nt_p	Number of fractional bits used by N_t
NI	Geometric coder maximum number of iterations

REFERENCES

[1] S. J. Visser, A. S. Dawood, and J. A. Williams, "FPGA based satellite adaptive image compression system," *J. Aerosp. Eng.*, vol. 16, no. 3, pp. 129–137, 2003.

[2] Z. Cao, T. Zhang, M. Liu, and H. Luo, "Wavelet-supervision convolutional neural network for restoration of JPEG-LS near lossless compression image," in *Proc. IEEE Asia Conf. Inf. Eng. (ACIE)*, Jan. 2021, pp. 32–36.

[3] M. R. Lone, "A high speed and memory efficient algorithm for perceptually-lossless volumetric medical image compression," *J. King Saud Univ.-Comput. Inf. Sci.*, Apr. 2020. [Online]. Available: https://service.elsevier.com/app/answers/detail/a_id/22801/supporthub/sciencedirect/

[4] G. Placidi, "Adaptive compression algorithm from projections: Application on medical greyscale images," *Comput. Biol. Med.*, vol. 39, no. 11, pp. 993–999, Nov. 2009.

[5] Q. Al-Shebani, P. Premaratne, P. J. Vial, and D. J. McAndrew, "The development of a clinically tested visually lossless image compression system for capsule endoscopy," *Signal Process., Image Commun.*, vol. 76, pp. 135–150, Aug. 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092359651830290X

[6] Y. Kim, "Real-time medical imaging with multimedia technology," in *Proc. IEEE Eng. Med. Biol. Soc. Region Int. Conf. Inf. Technol. Appl. Biomed. (ITAB)*, Sep. 1997, pp. 33–37.

[7] M. Yang and N. Bourbakis, "An overview of lossless digital image compression techniques," in *Proc. 48th Midwest Symp. Circuits Syst.*, vol. 2, 2005, pp. 1099–1102.

[8] M. J. Weinberger, G. Seroussi, and G. Sapiro, "From LOGO-I to the JPEG-LS standard," in *Proc. Int. Conf. Image Process.*, vol. 4, 1999, pp. 68–72.

[9] L. Chen, L. Yan, H. Sang, and T. Zhang, "High-throughput architecture for both lossless and near-lossless compression modes of LOCO-I algorithm," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 29, no. 12, pp. 3754–3764, Dec. 2019.

[10] M. Ferretti and M. Boffadossi, "A parallel pipelined implementation of LOCO-I for JPEG-LS," in *Proc. 17th Int. Conf. Pattern Recognit. (ICPR)*, vol. 1, 2004, pp. 769–772.

[11] M. Klimesh, V. Stanton, and D. Watola, "Hardware implementation of a lossless image compression algorithm using a field programmable gate array," *Mars*, vol. 4, no. 4.69, pp. 5–72, 2001.

[12] M. Ferretti and M. Boffadossi, "A parallel pipelined implementation of LOCO-I for JPEG-LS," in *Proc. 17th Int. Conf. Pattern Recognit. (ICPR)*, vol. 1, 2004, pp. 769–772.

[13] P. Merlino and A. Abramo, "A fully pipelined architecture for the LOCO-I compression algorithm," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 7, pp. 967–971, Jul. 2009.

[14] X. Li, X. Chen, X. Xie, G. Li, L. Zhang, C. Zhang, and Z. Wang, "A low power, fully pipelined JPEG-LS encoder for lossless image compression," in *Proc. IEEE Multimedia Expo Int. Conf.*, Jul. 2007, pp. 1906–1909.

[15] L.-J. Kau and S.-W. Lin, "High performance architecture for the encoder of JPEG-LS on SOPC platform," in *Proc. SiPS*, Oct. 2013, pp. 141–146.

[16] A. Kiely and M. Klimesh, "The ICER progressive wavelet image compressor," Jet Propuls. Lab., Pasadena, CA, USA, IPN Prog. Rep. 42-155, 2003, pp. 1–46, vol. 42, no. 155.

[17] J. Duda, "Asymmetric numeral systems," *CoRR*, vol. abs/0902.0271, pp. 1–47, May 2009. [Online]. Available: http://arxiv.org/abs/0902.0271

[18] J. Duda, "Asymmetric numeral systems: Entropy coding combining speed of Huffman coding with compression rate of arithmetic coding," *CoRR*, vol. abs/1311.2540, pp. 1–24, Nov. 2013. [Online]. Available: http://arxiv.org/abs/1311.2540

[19] J. Duda, K. Tahboub, N. J. Gadgil, and E. J. Delp, "The use of asymmetric numeral systems as an accurate replacement for Huffman coding," in *Proc. Picture Coding Symp. (PCS)*, May 2015, pp. 65–69.

[20] *LOCO-ANS Repository*. Accessed: Jun. 27, 2021. [Online]. Available: https://github.com/hpcn-uam/LOCO-ANS

[21] *Digital Compression and Coding of Continuous Tone Still Images—Requirements and Guidelines*, document ITU-T T.81-ISO/IEC is 10918-1, ITU-T Recommendation T, 1993, vol. 81.

[22] *Portable Network Graphics (PNG)*. Accessed: Apr. 20, 2021. [Online]. Available: http://www.libpng.org/pub/png/libpng.html

[23] 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.

[24] S. Golomb, "Run-length encodings," *IEEE Trans. Inf. Theory*, vol. IT-12, no. 3, pp. 399–401, Jul. 1966.

[25] *Information Technology—Lossless and Near-Lossless Compression of Continuous-Tone Still Images: Extensions*, document ITU-T T. 870-ISO/IEC 14495- 21, ITU-T, 2003.

[26] X. Wu, N. Memon, and K. Sayood, *A Context-Based, Adaptive, Lossless/Nearly-Lossless Coding Scheme for Continuous-Tone Images*, document ISO/IEC JTC 1/SC 29/WG, 1995, vol. 1.

[27] X. Wu and N. Memon, "Context-based, adaptive, lossless image coding," *IEEE Trans. Commun.*, vol. 45, no. 4, pp. 437–444, Apr. 1997.

[28] *Information Technology-Lossless and Near-Lossless Compression of Continuous-Tone Still Images: Baseline*, document ITU-T T. 87-SO/IEC 14495-1, ITU-T, Jun. 1998.

[29] J. Rissanen, "Generalized Kraft inequality and arithmetic coding," *IBM J. Res. Develop.*, vol. 20, no. 3, pp. 198–203, May 1976.

[30] S. M. Najmabadi, Z. Wang, Y. Baroud, and S. Simon, "High throughput hardware architectures for asymmetric numeral systems entropy coding," in *Proc. 9th Int. Symp. Image Signal Process. Anal. (ISPA)*, Sep. 2015, pp. 256–259.

[31] S. M. Najmabadi, H. S. Tungal, T.-H. Tran, and S. Simon, "Hardware-based architecture for asymmetric numeral systems entropy decoder," in *Proc. Conf. Design Archit. Signal Image Process. (DASIP)*, Sep. 2017, pp. 1–6.

[32] S. M. Najmabadi, T.-H. Tran, S. Eissa, H. S. Tungal, and S. Simon, "An architecture for asymmetric numeral systems entropy decoder—A comparison with a canonical Huffman decoder," *J. Signal Process. Syst.*, vol. 91, no. 7, pp. 805–817, Jul. 2019.

[33] Rawzor. *Rawzor Test Images*. Accessed: Apr. 20, 2021. [Online]. Available: http://imagecompression.info/test_images/

[34] *Libjpeg Implementation*. Accessed: Jan. 7, 2021. [Online]. Available: https://github.com/thorfdbg/libjpeg

[35] N. Merhav, G. Seroussi, and M. J. Weinberger, "Coding of sources with two-sided geometric distributions and unknown parameters," *IEEE Trans. Inf. Theory*, vol. 46, no. 1, pp. 229–236, Jan. 2000.

[36] R. F. Rice, "Some practical universal noiseless coding techniques," Jet Propuls. Lab., Pasadena, CA, USA, Tech. Rep. 76-22, 1979.

[37] *Charls Implementation of JPEG-LS*. Accessed: Apr. 20, 2021. [Online]. Available: https://github.com/team-charls/charls

[38] *CALIC Implementation*. Accessed: Jun. 15, 2021. [Online]. Available: https://github.com/Tobi-Alonso/gcif/tree/master/refs/calic

[39] *Information Technology—JPEG 2000 Image Coding System: Core Coding System*, document ITU-T T.800 | ISO/IEC 15444-1, ITU-T, Jun. 2019.

[40] *JPEG2000 Implementation*. Accessed: Jun. 9, 2021. [Online]. Available: https://github.com/uclouvain/openjpeg

[41] *Information Technology—JPEG 2000 Image Coding System: High-Throughput JPEG 2000*, document ITU-T T.814 | ISO/IEC 15444-15, ITU-T, Jun. 2019.

[42] *High Throughput JPEG2000 Implementation. Tag: 0.7.3*. Accessed: Jun. 9, 2021. [Online]. Available: https://github.com/aous72/OpenJPH

[43] *Webp Homepage*. Accessed: Jun. 15, 2021. [Online]. Available: https://developers.google.com/speed/webp

[44] *Libwebp Implementation*. Accessed: Jun. 15, 2021. [Online]. Available: https://github.com/webmproject/libwebp

[45] *Libwebp2 Implementation*. Accessed: Jun. 15, 2021. [Online]. Available: https://chromium.googlesource.com/codecs/libwebp2

[46] *Overview of JPEG XL*. Accessed: Jun. 24, 2021. [Online]. Available: https://jpeg.org/jpegxl/index.html

[47] A. Rhatushnyak, J. Wassenberg, J. Sneyers, J. Alakuijala, L. Vandevenne, L. Versari, R. Obryk, Z. Szabadka, E. Kliuchnikov, I.-M. Comsa, K. Potempa, M. Bruse, M. Firsching, R. Khasanova, R. van Asseldonk, S. Boukortt, S. Gomez, and T. Fischbacher, "Committee draft of JPEG XL image coding system," 2019, *arXiv:1908.03565*. [Online]. Available: https://arxiv.org/abs/1908.03565

[48] *JPEG XL Reference Software*. Accessed: May 28, 2021. [Online]. Available: https://gitlab.com/wg1/jpeg-xl

[49] *Challenge on Learned Image Compression*. Accessed: Jun. 15, 2021. [Online]. Available: http://compression.cc/tasks/

[50] *CLIC Images Subset*. Accessed: Jun. 15, 2021. [Online]. Available: https://drive.google.com/drive/folders/1wMgmjf54iN46dVihvMnHhGk8oQT7a8Nd

[51] A. V. Lotov and K. Miettinen, "Main terminology and notations used," in *Multiobjective Optimization*. Berlin, Germany: Springer, 2008, ch. 7, pp. 10–11.

[52] J. Alakuijala, J. Sneyers, L. Versari, and J. Wassenberg. *JPEG White Paper: JPEG XL Image Coding System*. Accessed: Jun. 24, 2021. [Online]. Available: http://ds.jpeg.org/whitepapers/jpeg-xl-whitepaper.pdf

[53] T. Alonso, M. Ruiz, Á. L. Garcia-Arias, G. Sutter, and J. E. L. de Vergara, ''Submicrosecond latency video compression in a low-end FPGA-based system-on-chip,'' in *Proc. 28th Int. Conf. Field Program. Log. Appl. (FPL)*, Aug. 2018, pp. 355–3554.

[54] T. Liebchen, T. Moriya, N. Harada, Y. Kamamoto, and Y. A. Reznik, ''The MPEG-4 audio lossless coding (ALS) standard-technology and applications,'' in *Proc. 119th AES Conv.*, 2005, pp. 1–14.

[55] *FLAC—Free Lossless Audio Codec*. Accessed: Apr. 20, 2021. [Online]. Available: https://xiph.org/flac/

TOBÍAS ALONSO received the degree in electronic engineering from Universidad Nacional de San Juan (UNSJ), Argentina, in 2017. He is currently pursuing the Ph.D. degree with Universidad Autónoma de Madrid, Spain. He was a Teaching Assistant with UNSJ. He is also a Research and Teaching Assistant with Universidad Autónoma de Madrid. His research interests include FPGA hardware design for high-speed networks, algorithm acceleration, and development of embedded systems.

GUSTAVO SUTTER (Member, IEEE) received the M.S. degree in computer science from State University UNCPBA, Tandil, Buenos Aires, Argentina, in 1997, and the Ph.D. degree from the Autonomous University of Madrid, Spain, in 2005. He has been a Professor with UNCPBA. He is currently a Professor with Universidad Autónoma de Madrid. He is the author of 3 books and more than 100 international articles and communications. His research interests include FPGA design, digital arithmetic, development of embedded systems, and high-performance computing.

JORGE E. LÓPEZ DE VERGARA (Senior Member, IEEE) received the M.Sc. and Ph.D. degrees in telecommunication engineering from Universidad Politécnica de Madrid, Spain, in 1998 and 2003, respectively. He is currently an Associate Professor with Universidad Autónoma de Madrid, Spain, and a Founding Partner of Naudit HPCN, a company devoted to high performance traffic monitoring and analysis. He has coauthored more than 100 scientific articles on this topic. His research interests include network and service management and monitoring.