A Relaxed Half-Stochastic Iterative Decoder for LDPC Codes

François Leduc-Primeau, Saied Hemati, Warren J. Gross, and Shie Mannor
Department of Electrical and Computer Engineering, McGill University, Montreal, QC H3A 2A7, Canada.
francois.leduc-primeau@mail.mcgill.ca, {saied.hemati, warren.gross, shie.mannor}@mcgill.ca

Abstract—This paper presents a Relaxed Half-Stochastic (RHS) low-density parity-check (LDPC) decoding algorithm that uses some elements of the sum-product algorithm (SPA) in its variable nodes, but maintains the low-complexity interleaver and check node structures characteristic of stochastic decoders. The algorithm relies on the principle of successive relaxation to convert binary stochastic streams to a log-likelihood ratio (LLR) representation. Simulations of a (2048, 1723) RS-LDPC code show that the RHS algorithm can outperform 100-iterations floating-point SPA decoding. We describe approaches for low-complexity implementation of the RHS algorithm. Furthermore, we show how the stochastic nature of the belief representation can be exploited to lower the error floor.

I. INTRODUCTION

Low-Density Parity-Check (LDPC) codes are a family of capacity-approaching error-correction codes originally proposed by Gallager in the early 1960’s [1] and then more recently rediscovered by MacKay [2]. LDPC decoding exhibits a high level of parallelism that scales with the code length. LDPC codes are therefore particularly interesting for high throughput applications. Practical LDPC codes typically have lengths upward of $10^3$ bits, and decoding circuits that fully exploit the available parallelism of the code become challenging to implement. One of the main difficulties resides with the interleaver that delivers messages between variable nodes and check nodes, because of the unusually high ratio of interconnections to logic induced by the topology of the code’s Tanner graph.

Stochastic decoding was introduced in [3], [4] as a low-complexity alternative to the sum-product algorithm (SPA) [5] that had the potential to achieve a similar performance. However, only the decoding of short codes resulted in compelling performance. In [6], the authors improved the stochastic decoding algorithm in order to achieve good decoding performance on larger codes. The decoder was refined again in [7], which also presented FPGA implementation results for practical codes.

Performance close to 64-iterations floating-point SPA was reported in [6] for codes of size 200 and 1024, but that performance was achieved with a maximum iteration count of 10K and 60K, respectively. The results reported in [7] reduce the upper limit to 700 iterations, but the performance is approximately 0.5 dB away from 32-iterations floating-point SPA (at a BER of $10^{-9}$).

A very attractive aspect of stochastic decoders is that they only require two wires per bidirectional edge of the Tanner graph, therefore bringing the circuit’s interconnect to its simplest expression. The check nodes are also extremely simple, containing only XOR gates. This structure makes it realistic to consider fully-parallel implementations of long codes. This is of particular advantage for implementing codes with high-degree check nodes, such as the (6,32)-regular RS-LDPC code used in the IEEE 802.3an 10-Gbps Ethernet standard. The check nodes for SPA and min-sum algorithm (MSA) decoders of this code have high-area implementations, even for bit-serial MSA decoders [8], [9]. In addition to the low complexity, another important advantage of stochastic decoders is the ability to add pipeline stages on some or all edges of the interleaver without affecting the error-correction performance [7]. This allows higher clock rates, and for an ASIC implementation can facilitate the layout and routing.

Motivated by the simple check node and interleaver structure, we sought a method for improving the bit-error-rate (BER) performance of stochastic decoders. In this paper, we introduce the Relaxed Half-Stochastic (RHS) decoding algorithm that processes the check node messages using stochastic decoding techniques, and the variable node messages using the SPA (and MSA) variable node update. We introduce a method for efficiently converting messages between the stochastic and probability (and log-likelihood-ratio) representations. In contrast to previous contributions [10]–[13] that represent probabilities by a fixed number of bits in a stochastic packet and have significant BER degradation, our algorithm is not packetized and operates on a bit-by-bit basis, updating probability messages in the variable nodes based on the principle of successive relaxation. Simulations show that the RHS algorithm significantly improves the BER performance, outperforming the floating-point sum-product algorithm.

We also show that the RHS algorithm has another advantage that stems from the use of stochastic streams. The random nature of the decoding process allows successful decoding of some received frames that initially resulted in a decoding error by restarting the decoder from the initial state, without increasing the iteration limit. This technique, called redecoding, has the effect of lowering the error floor.

The remainder of the paper is organized as follows. Section II summarizes the well-known sum-product algorithm, as well as the existing work on stochastic decoding. Section III introduces the RHS algorithm and the redecoding process and presents simulation results. Section IV addresses the implementation difficulties and introduces a low-complexity RHS decoder. Finally, Section V concludes the paper.
II. BACKGROUND

A. The Sum-Product Algorithm

Iterative decoding algorithms are often defined on the Tanner graph representation of the codes. A Tanner graph is a bipartite graph where the first type of nodes, called *variable nodes*, represent the codeword bits and the second type of nodes, the *check nodes*, represent the parity-check equations that form the constraints of the code. An edge extends from variable node \( i \) to check node \( j \) if the \( i \)-th codeword bit is included in the \( j \)-th parity-check equation, i.e. \( H_{j,i} = 1 \).

The sum-product algorithm operates by having the variable nodes and check nodes iteratively exchange messages that reflect each node’s local belief about their true value. SPA is the optimal decoding for linear block codes that have a cycle-free Tanner graph. When that condition is not satisfied, as is the case with all practical LDPC codes, SPA still provides excellent performance, which is however not optimal. The belief (or likelihood) values that form the messages can be expressed in a variety of representations, but in practice the log-likelihood ratio (LLR) representation is most often used. The LLR is defined in terms of a probability as 

\[
\Lambda = \log(\frac{q}{p})
\]

...be expressed in a variety of representations, but in practice the log-likelihood ratio (LLR) representation is most often used. The LLR is defined in terms of a probability as \( \Lambda = \log(\frac{q}{p}) \), where \( q = 1 - p \). For the case of BPSK modulation over an AWGN channel, the a-priori LLR \( \Lambda_i \) is obtained from the channel value \( y_i \) using (1), where \( N_o = 2\sigma^2 \) is the channel noise power. At each decoding iteration, the messages sent from variable and check nodes are defined respectively by (2) and (3), where \( i \) is the variable node index, \( j \) the check node index, \( V_{i,j} \) the set of check nodes connected to variable node \( i \) (excluding check node \( j \)) and \( C_{j,i} \) the equivalent for check nodes. The notation \( \mu' \) refers to the next message \( \mu \), and similarly for \( \nu' \).

\[
\Lambda_i = \frac{4y_i}{N_o} \quad (1)
\]

\[
\mu'_{i\rightarrow j} = \Lambda_i + \sum_{l \in V_{i,j}} \nu_{l\rightarrow i} \quad (2)
\]

\[
\nu'_{j\rightarrow i} = 2 \tanh^{-1}\left( \frac{\prod_{l \in C_{j,i}} \tanh\left( \frac{\mu_{l\rightarrow j}}{2} \right)}{2} \right) \quad (3)
\]

The algorithm terminates when a valid codeword is found, or when a pre-determined number of iterations has been reached.

B. Stochastic Decoding

Stochastic decoders represent the likelihood information in the form of binary stochastic streams, that is random Bernoulli sequences of arbitrary length, and the information contained in a sequence is its first order statistics. This representation becomes useful when the stream’s mean function is interpreted as a probability. In that case, low-complexity variable nodes and check nodes can be designed [3]. Despite the simple implementation, the stochastic stream representation can have arbitrary precision, and for this reason stochastic decoding was proposed as a low-complexity alternative to SPA decoding.

Considering the check node function of a degree-3 node (two inputs and one output), let the expected value of the current input bits be \( p_0 \) and \( p_1 \). Applying the XOR function on the inputs, the expected value of the output is \( p_{out} = p_0q_1 + q_0p_1 \) (where \( q_1 = 1 - p_1 \)), which is the exact check node function of SPA in the probability domain [5]. Since the function does not reference any previous values of the input streams, it remains valid for non-stationary streams. The function is easily extended to higher degree nodes by combining together multiple degree-3 nodes [3].

A number of functions have been proposed for the variable node operation [3], [6], [10]. The variable node function given below was introduced in [3]. This function is defined for a variable node of degree 3 (inputs \( \{a, b\} \), output \( c \)) and has only 1 state bit. It can easily be shown that if each input stream is assumed stationary, the expected value of the output stream represents the desired probability.

\[
c_i = \begin{cases} a_i & \text{if } a_i = b_i, \\ c_{i-1} & \text{otherwise.} \end{cases} \quad (4)
\]

It is shown in [6] that this approach suffers from an early error floor at BER=\(10^{-3} \) when decoding a code of length 200. Two improvements that resolve this early floor were introduced in [6]. First, noise-dependent scaling consists in scaling down the a-priori channel likelihoods in order to increase the switching activity in the stochastic streams. The variable node function is also modified to include memory, referred to as an edge-memory. To this end, the authors define regenerative bits as the output of (4) in the case of equality of the inputs. The memory is used much in the same way as the single state bit of (4), but stores many regenerative bits. When the current output \( c_i \) is not regenerative, a bit is instead sampled randomly from the edge-memory.

In [14], tracking forecast memories (TFMs) are introduced to extract the mean of regenerative bits, as a replacement for edge-memories. They have the advantage of a lower hardware complexity, but do not improve the performance of the decoder, which remains approximately 0.5 dB away from 32 iterations floating-point SPA. A similar function inspired by the TFM is used in the RHS algorithm in a different context. This is presented in the following section.

III. THE RHS ALGORITHM

A. Algorithm

As was noted in [6] and [14], it is likely that the gap between the performance of floating-point SPA and of stochastic decoding is due to the variable node implementation. The RHS algorithm improves the accuracy of the variable node operation by extracting the information in the stochastic stream and performing exact operations in the LLR domain. By exact operations, we mean that the variable node function of the SPA is used directly. The rest of the decoder remains stochastic, and information is exchanged between variable and check nodes in the form of stochastic streams. The RHS variable node is constructed by extending the SPA variable node to handle incoming stochastic streams and convert them to LLR values, and to be able to generate the stochastic streams at the output. The resulting functionality of the variable node is illustrated in Fig. 1, which is described in the following paragraphs.
1) Converting Stochastic Streams to LLR Values: At the inputs of the variable node, stochastic streams are converted to LLR values. First, the probability value that is embedded in the stochastic stream must be estimated, and second, that probability must be converted to the LLR domain. The same function used in [14] for TFM at the output of the stochastic variable node can be used here at the input. This function is also known as an exponential moving average and is a well-known first-order tracking filter. It is expressed as

\[ p_i = (1 - \beta)p_{i-1} + \beta b_i. \]  

(5)

It generates an estimate of the current probability \( p_i \) by updating the previous estimate \( p_{i-1} \) according to the received bit \( b_i \). \( \beta \) controls the sensitivity of the tracker to new information. We will refer to it as the relaxation factor, because this function is the one used in the successive relaxation (SR) algorithm [15]. In the context of iterative decoding, successive relaxation consists in gradually changing the output messages from one iteration to the next (using (5), and interpreting \( b_i \) as a general input), instead of completely replacing the previous outputs with the new ones. It was shown in [15] that SR can improve the performance of iterative decoding algorithms. In the RHS algorithm, the relaxation operation is applied at the input of the variable node. This is equivalent to applying it at the output if the output is also in the probability domain. Therefore it can be assumed that the RHS algorithm applies successive relaxation in the probability domain.

The relaxation factor \( \beta \) controls the weight of the current stochastic bit in determining the current probability estimate. Because it controls the sensitivity of the probability estimate, it sets a bound as to how fast the entire iterative decoder can converge to a codeword. In [15] the authors note that reducing \( \beta \) while increasing the iteration limit has the effect of improving the decoding performance, until it reaches the optimal continuous-time performance. They also note that under a limited number of iterations, there exists an optimal \( \beta \) in terms of performance. Our experimental results tend to confirm these observations. We also noted that it is possible to optimize the average response of the decoder to obtain a higher throughput at the same signal-to-noise ratio (SNR) and frame error rate, by changing the value of \( \beta \) after a number of cycles.

2) Generating Output Streams: The output messages in the LLR domain are computed by converting all input probabilities to LLR values and performing the necessary additions, as in (2). The outputs must then be converted to stochastic streams. Each bit of a stochastic stream is generated by comparing the desired expected value with a uniform random number. Depending on the result of the comparison, a 1 or a 0 is output. The expected value of the stream being in the probability domain, one way to generate an output bit is to convert the output LLR to the probability domain, and then compare that probability value with a uniform random number. Equivalently, the random number can be converted to the LLR domain, and compared with the output LLR. The latter is preferred because many stochastic output bits can be generated from the same random number. In fact, it has been reported in [7] that only a small number of independent random number generators (RNGs) are needed in a complete stochastic decoder.

B. Redecoding

The stochastic streams that are used to exchange information in the decoder are random sequences, and the state of the decoder at any given time is dependent on the specific realizations of those random sequences. It follows that the ultimate success or failure of the decoder is a random experiment, conditioned by the decoding algorithm. For this reason, a set of received channel values that cause the decoder to fail might be decoded successfully on a second attempt. It is also possible that the failed frame would have been successfully decoded had we let the decoder run for more iterations.

It is helpful to think of the RHS decoding algorithm as a randomized optimization algorithm [17]. If a given optimization algorithm operating on a given problem is equally efficient in all its possible states, then no trajectory through the state-space is better than another, and nothing can be gained from running a second randomized experiment, as opposed to running the initial experiment for more iterations. If however the algorithm is more efficient in some (assumed unknown) parts of its state-space, then it can be more cost effective to run many short randomized experiments instead of a long one. Furthermore, if the state-space contains local optima, some trajectories will lead to dead ends where the algorithm stops making progress completely. Local optima were shown in [18] to be responsible for error floors in the decoding of LDPC codes, and the term trapping set was introduced to describe the structures in the code graph responsible for those local optima.

Simulations have shown that redecoding can greatly improve the error rate performance in the error floor region. Those results are presented below.

C. Simulation Results for the RHS Algorithm

The simulation framework is composed of a random source with an encoder, BPSK modulation/demodulation and an AWGN channel. All the simulations are performed with a maximum of 1000 decoding cycles using early termination. A decoding cycle corresponds to 1 bit being processed by the variable nodes, sent to the check nodes and sent back to the variable nodes. We note that a decoding cycle in a stochastic decoding algorithm is not equivalent to an iteration in a message passing algorithm such as SPA [7]. The code used.
for the simulations is a (6, 32)-regular RS-based LDPC code of size 2048 and rate 0.84, which has a girth of 6. This code was the one selected in the recent IEEE 802.3an standard for 10-Gbps Ethernet over CAT6 cables [19]. The decoder presented in this section uses floating-point representations.

The effects of the relaxation factor $\beta$ is illustrated by the settling curves in Fig. 2. The curves show the frame error rate of the decoder as a function of the maximum number of decoding cycles (DCs) at 4.2 dB. When the maximum number of DCs is between approximately 115 and 740, the best performance is obtained with $\beta = 1/16$, while $\beta = 1/32$ does a little bit better when the DC limit is above 740. On the other hand, the decoder converges faster on average for higher values of $\beta$, as shown by the initial part of the curves being further to the left. The average number of decoding cycles at 4.2 dB is 20 in the case of $\beta = 1/8$, and 30 for $\beta = 1/16$. By using $\beta = \{1/4, 1/8, 1/16\}$ in sequence over the course of the decoding process, this number can be reduced to 15 without any degradation in the decoding performance. The curve shown on Fig. 2 uses $\beta = 1/4$ for cycles $[0, 33]$, $\beta = 1/8$ for cycles $[34, 110]$, and then $\beta = 1/16$. Looking at the graph we can extrapolate that additional decoding cycles would provide little additional performance.

The BER performance of the RHS decoder is shown on Fig. 3, alongside 100-iterations floating-point SPA with 5-bit demodulation. The curve referred to as “ASPA” is the “offset approximate SPA” decoder with 4-bit quantization presented in [21]. Also in the figure is a 7-bit SPA implementation presented in [20], as well as a stochastic decoder designed according to the architecture presented in [7]. This decoder is simulated with 64-bit edge-memories and a limit of 1000 decoding cycles. In the waterfall region, at a BER of $10^{-8}$, the RHS decoder outperforms all the other decoders, but most importantly 100 iterations floating-point SPA. It is worth mentioning that simulations confirm that the RHS decoder matches the performance of optimized relaxed SPA when using a sufficiently small relaxation factor.

Fig. 4 shows that the redecoding technique can significantly lower the error floor of the RHS decoder. A solution to the error floor problem was also presented in [21], where a postprocessing technique was introduced that relies on prior characterization of the trapping sets of the code. The postprocessing operates by applying carefully chosen offsets to messages in the neighborhood of unsatisfied check nodes. The performance of this ASPA decoder with postprocessing is also shown on Fig. 4. At 4.5 dB, the performance achieved by the RHS decoder is equivalent to that obtained from the postprocessing technique of [21]. The error-floor of RHS with redecoding is lower than SPA and follows the same shape. At lower frame error rates, the observed trend indicates that the RHS decoder would be outperformed by the ASPA decoder with postprocessing. Note that similar techniques can also be applied to RHS and this will be explored in future work. However, the cost of the extra postprocessing hardware in general may not be acceptable for many applications.

Simulation results are presented with a maximum of 1000 cycles, but this limit can be reduced. Fig. 3 also shows a RHS simulation that uses a limit of 160 cycles, which is sufficient to match the performance of [21]. To compare the throughput of RHS and a MSA implementation such as [21], we note that at 5.5 dB the average convergence time for RHS is 9.6 clock cycles (2.4 iterations, 4 clocks/iteration), while it is 19.5 clock cycles for MSA (1.5 iteration, 13 clocks/iteration). Hence, a solution to error floor problem was also presented in [21], where we can extrapolate that additional decoding cycles provide little additional performance.

**IV. RHS DECODER IMPLEMENTATION**

We now address the main challenges in implementation of the functionality described in the previous section. The discussion only concerns the implementation of the variable node, since the interleaver and check nodes are straightforward. We will present the implementation in three parts: the front-end that converts stochastic streams to LLR values, the central
SPA summation circuit, and the back-end that generates the outgoing stochastic streams.

A. Stochastic Stream Tracking

On the front-end of the variable node, each incoming stochastic stream must be converted to a LLR value. One way to do this is to track the probability value contained in the stream, and then convert that probability to the LLR domain. However, in this context there is a more interesting alternative, which is to design a tracking mechanism that operates directly in the LLR domain. The exponential moving average presented in Equation (5) becomes non-linear in the LLR domain, but the summation in the variable node is the same as the SPA summation circuit, and the back-end that generates the LH. The linear fit was optimized so that the expected value is expressed in the LLR domain, the random number is converted to the LLR domain to perform the comparison. In practice, the random numbers will have a finite precision, and this defines the effective range of the random number generator (RNG), which is the range associated with the stochastic stream. The smallest probability that can be generated by the RNG is given by $1/2^n$, where $n$ is the bit width of the RNG. The upper bound of the LLR range is then $\log 2^n + \log(1 - 2^{-n})$. To cover a range of at least $[-8, 8]$, the RNG needs 12 bits and the actual range is approximately $[-8.3, 8.3]$.

The random values in the probability domain can be converted to the LLR domain using a look-up table (LUT), but indexing the table directly with the probability value results in a table with $2^{12}$ entries, which is impractical. The next section shows that LLR values can be quantized on 4 or 5 bits. In the case of a 4-bit quantization, there are only $2^3$ positive output values. Starting from this 8 entries LUT, we wish to devise an addressing circuit that takes a probability value as input. The LLR domain being logarithmic, taking the log of the probability value can provide a good indexing mechanism. Considering a probability value $p$ expressed in binary form and satisfying $0 < p \leq 1/2$, the integer part of the log in base 2 is easily obtained by counting the number of leading zeros before the first one. This generates 12 indexes, which can be further reduced to 8 by OR’ing index lines that map to the same quantized LLR value. If $p > 1/2$, the same technique can be applied to $1 - p$ while changing the sign of the resulting LLR value. By considering the random number as composed of a sign bit and of a probability value in the range $0 < p \leq 1/2$, the subtraction can be avoided. Table I illustrates the LUT for the case of a 12-bit RNG and 4-bit LLR. The three dots indicate that all the other bits are “don’t cares”.

C. Stochastic Stream Generation

Stochastic bits are generated by comparing an expected value to a random number. The mean function of stochastic streams is in the probability domain, and therefore the RNG must generate uniform random numbers in the probability domain. The random number is converted to the LLR domain to perform the comparison. In practice, the random numbers will have a finite precision, and this defines the effective range of the random number generator (RNG), which is the range associated with the stochastic stream. The smallest probability that can be generated by the RNG is given by $1/2^n$, where $n$ is the bit width of the RNG. The upper bound of the LLR range is then $\log 2^n + \log(1 - 2^{-n})$. To cover a range of at least $[-8, 8]$, the RNG needs 12 bits and the actual range is approximately $[-8.3, 8.3]$.

The random values in the probability domain can be converted to the LLR domain using a look-up table (LUT), but indexing the table directly with the probability value results in a table with $2^{12}$ entries, which is impractical. The next section shows that LLR values can be quantized on 4 or 5 bits. In the case of a 4-bit quantization, there are only $2^3$ positive output values. Starting from this 8 entries LUT, we wish to devise an addressing circuit that takes a probability value as input. The LLR domain being logarithmic, taking the log of the probability value can provide a good indexing mechanism. Considering a probability value $p$ expressed in binary form and satisfying $0 < p \leq 1/2$, the integer part of the log in base 2 is easily obtained by counting the number of leading zeros before the first one. This generates 12 indexes, which can be further reduced to 8 by OR’ing index lines that map to the same quantized LLR value. If $p > 1/2$, the same technique can be applied to $1 - p$ while changing the sign of the resulting LLR value. By considering the random number as composed of a sign bit and of a probability value in the range $0 < p \leq 1/2$, the subtraction can be avoided. Table I illustrates the LUT for the case of a 12-bit RNG and 4-bit LLR. The three dots indicate that all the other bits are “don’t cares”.

D. Simulation Results for the RHS Implementation

This section illustrates some possible tradeoffs with respect to the error correction performance and the implementation
complexity. Two parameters are used to control the implementation’s precision: the number of bits used to represent LLR values in the LLR tracker, and the number of bits used in the rest of the variable node. The LLR tracker has the role of an estimator for the stochastic stream, and therefore its precision has an impact on the precision of the information received through the stochastic stream. In the rest of the variable node, the same representation is used for LLR values in the adder circuit and for the LLR-domain random number. The representations used for the simulations are 9-bit or 12-bit for the LLR tracker, and 4 or 5-bit for the other LLR values. Fig. 5 shows that the performance of the proposed circuit implementation is very close to the ideal case, even though LLR values are quantized on 4 or 5 bits. At a BER of $10^{-8}$, the (12, 5) implementation (12-bit LLR tracker and 5-bit quantization in the rest of the variable node) has a loss of only 0.04 dB. The throughput is as good as in the non-simplified decoder for all the implementation cases.

V. CONCLUSION

In this paper, we presented a stochastic decoding algorithm for LDPC codes that has two major novel aspects. First, variable nodes use a method similar to successive relaxation to estimate the current value of incoming stochastic streams. Together with the use of LLR domain operations, this creates a very accurate stochastic variable node, as shown by the excellent error correction performance obtained in the waterfall region. Second, we showed how the random convergence behavior of the decoder could be exploited to lower the error floor that limits the decoding performance at high SNRs. A simplified fixed-point implementation was then presented, with a performance close to that of the ideal algorithm.

Despite the increased performance, the RHS decoder maintains the advantages of other stochastic decoders concerning the implementation of high-speed fully-parallel decoders. The minimal interlaver, the simple check nodes and the ability to arbitrarily pipeline the interlaver are big advantages in that context.

We believe that random convergence properties could play an important role in improving existing iterative decoding algorithms to overcome structural deficiencies in the codes. The fact that those properties are a natural part of all stochastic decoders makes them even more attractive for high-speed low BER applications.

ACKNOWLEDGEMENT

Authors wish to thank the anonymous reviewers for their valuable comments, WestGrid, RQCHP, and CLUMEQ for providing computing resources, and NSERC for funding.

REFERENCES