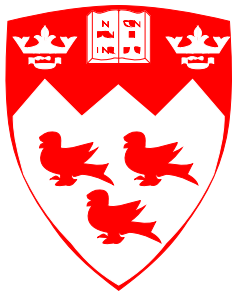
Stochastic Decoding of LDPC Codes over GF(q)

Gabi Sarkis



Department of Electrical and Computer Engineering

McGill University

Montréal, Québec

August 2009

A thesis submitted to McGill University in partial fulfilment of the requirements of the degree of Master of Engineering

ACKNOWLEDGEMENTS

I would like to thank Professor Warren J. Gross for all the guidance and advice he has given and continues to give me. I also would like to thank Professor Mannor for his continued support, François, Saied, and Ali for all the help they provided throughout my research. I am grateful to all my friends in Montréal for all the great times we had together. Thank you, Dani, Marwan, François, and all the others who make life here fun and interesting. Thank you Nicole for always being by my side. I want to thank the Chahines and Bannas for being my family away from home. I will always be grateful to Professor Lu of Purdue University for guiding me and helping me choose my path. Finally, I want to thank my parents and brother without whose support nothing would be possible.

ABSTRACT

Non-binary LDPC codes have been shown to outperform commonly used codes for many communications and storage channels. Currently proposed non-binary decoder architectures have very high complexity for high-throughput implementations and sacrifice error-correction performance to maintain realizable complexity. In this work, we present an alternative decoding algorithm based on stochastic computation that has a very simple implementation and minimal performance loss when compared to the sum-product algorithm. We demonstrate the performance of the algorithm when applied to a GF(16) code and provide details of the hardware resources required for an implementation with synthesis results for the decoder nodes. We also utilize two methods, originally used in binary stochastic decoding, to improve the non-binary decoder throughput and decrease its maximum latency.

ABRÉGÉ

Il a été démontré pour plusieurs types de canaux de télécommunication et de stockage que les codes LDPC non-binaires ont une meilleure performance que les codes les plus souvent utilisés. Par contre, les architectures de décodeur à haut débit qui ont été proposées jusqu'à maintenant sont très complexes, et sacrifient la performance de correction d'erreur de façon à maintenir une complexité acceptable. La présente thèse décrit un nouvel algorithme de décodage qui utilise une représentation stochastique des données pour arriver à une implémentation très simple, et souffrant de très peu de perte de performance par rapport à l'algorithme somme-produit. Nous rapportons la performance de l'algorithme appliqué à un code de GF(16), et présentons le détail des ressources matérielles requises à l'implémentation, accompagnés de résultats de synthèse circuit pour les noeuds du décodeur. Finalement, nous utilisons deux méthodes originellement développées pour les décodeurs stochastiques binaires pour améliorer le débit du décodeur non-binaire, et diminuer sa latence maximale.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

1.1 Motivation

With wireless communications and broadband Internet access becoming commonplace nowadays, we have come to expect fast and reliable methods for transferring data. To meet this expectation, the entire communication system, from end-user portable devices to the fiber-optic Internet backbone and the storage systems in data centers, must be resilient to errors introduced during data transmission and storage. Such a requirement necessitates that the data include structured redundancy so that when errors occur during transmission, the system can correct them and data corruption would be avoided.

Natural languages contain much redundancy and structure therefore it is possible to infer the intended word written even if it includes spelling error. For example if the word "stochastc" were to be encountered, a reader would note the spelling error and correct the word so that it is "stochastic". Digital data does not contain such structured redundancy, and therefore, contains no inherent protection against errors.

Error control coding (ECC) is a method for providing the required data protection. It adds structured redundant information to the data at the encoding stage before transmission. The resulting coded data (codeword) is sent. The receiver decodes the received data using knowledge of the code structure to correct any errors encountered.

ECC is so prevalent that it can be found in places ranging from spaceships approaching the limits of the Solar System, to the digital music players in one's pocket.

One ECC scheme that has been shown to have excellent performance approaching the theoretical limits is binary low-density parity-check (LDPC) coding [1]. A more powerful version of these codes can be obtained by grouping bits of the original message into symbols, similar to what is done in Reed-Solomon coding. The symbol-based codes are called LDPC codes over GF(q), or non-binary codes.

While non-binary LDPC codes have been shown to have performance exceeding that of most other codes over a variety of channels [2, 3, 4] , their decoding algorithms are extremely complex, prohibiting their widespread use. As such, there exists a need for a practically implementable decoding algorithm for LDPC codes over GF(q).

1.2 Contribution

In this thesis, we present a novel algorithm for decoding LDPC codes over GF(q) that has a lower complexity than other algorithms currently discussed in literature. To achieve this simpler design, binary stochastic decoding was generalized to function over GF(q). Simulations were performed and showed the proposed algorithm's performance matched the best performance in literature.

An architecture which implements the algorithm was developed, and modifications to enhance the decoder performance and further reduce its complexity were investigated.

1.3 Thesis Plan

Chapter 2 starts by providing background information on binary and non-binary LDPC decoding and reviewing the algorithms for decoding LDPC codes over GF(q) currently in literature. Our algorithm is presented in Chapter 3 which also includes a complexity analysis. Parts of this chapter were presented in the IEEE International Conference on Communications [5]. The corresponding implementation is shown with synthesis results in Chapter 4.

Further enhancements to increase the decoder throughput are presented in Chapter 5.

Chapter 2

Literature Review

2.1 Binary LDPC Codes

Binary low-density parity-check (LDPC) codes are linear block codes characterized by a sparse $M \times N$ parity check matrix H. They were first introduced by Gallager [6] in 1963. However, due to their encoding and decoding complexity, these codes were ignored for over 30 years until MacKay revived interest in them in the late 90's. MacKay [7] proved that LDPC codes had excellent performance approaching the Shannon limit for large block lengths and rivaling the turbo codes of Goff *et al.* [8]. Furthermore, he classified LDPC codes into regular and irregular codes and showed that irregular codes perform better. A regular code's H-matrix has constant row and column weights; while an irregular code has no such constraint.

As noted in [9], LDPC codes are block codes, and are encoded using a generator matrix G which satisfies $HG^T = 0$. A codeword is generated from a message u using $x = uG$. To check if a received vector y is a valid codeword, the syndrome vector z is computed using $z = yH^T$. If $z = 0$, then y is a valid codeword; otherwise, the decoder

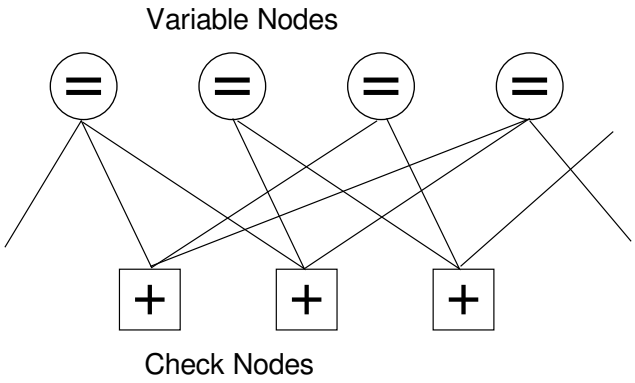


Figure 2–1: Tanner graph of a binary code

attempts to correct it. Decoding of LDPC codes is done via a belief propagation algorithm originally introduced by Gallager and later refined by MacKay.

2.1.1 Binary LDPC Decoding

Most LDPC decoders use bipartite graphs, called a Tanner graph, as a model when implementing the belief propagation algorithm of [6] and [7]. Such graphs are composed of two types of nodes: variable, also called equality, nodes and check nodes. Figure 2–1 is an example of a Tanner graph section.

The received vector bits y correspond to the variable nodes, and the computations of the syndrome bits, also called check constraints, to the check nodes. A connection between a variable node v and a check node c is made if the corresponding parity-check matrix element $h = H[c, v]$ is non-zero. Therefore, the weight of a row in H determines the number of variable nodes connected to the corresponding check node, which is called the check node degree d_c. Similarly the column weight determines the variable node degree d_v. Connected nodes exchange messages until all the checks are satisfied or a certain number of iterations has been reached.

The decoder messages can be computed exchanged using various method, the most direct of which is called the sum-product algorithm (SPA) and was implemented by [7] for channels with independent noise samples. In this algorithm, the messages are two element probability mass function (PMF) vectors such as $P = [p(0), p(1)]$, where $p(0)$ is the probability of the bit being 1 and $p(0)$ is the probability of it being 1. For labeling messages we following the notation used in [10]: messages from variable node v to check node c are denoted U_{vc}, and messages from c to v are denoted as V_{cv}.

The first step in the SPA is to compute a likelihood vector $L = [l(0),\ l(1)]$ for every received bit i and use it initialize the corresponding variable node. $l(k) = P[x_i = k|y_i]$ is the likelihood of the transmitted bit being equal to k given y_i was received and its computation is dependent on the channel model and modulation scheme used. For example, for an additive white Gaussian noise (AWGN) channel with binary phase-shift keying (BPSK) the following is used [9]:

$$l(0) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y+a)^2}{2\sigma^2}}, \quad l(1) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(y-a)^2}{2\sigma^2}} \tag{2.1}$$

The iterative decoding process begins by computing the variable node update messages using [9]:

$$U_{vc} = L \times \prod_{i=1, i\neq c}^{d_v} V_{iv} \tag{2.2}$$

where $\times$ is a term-by-term product of vectors. Since each message U_{vp} is a PMF, it must be normalized such that $\sum_{i=0}^{1} U_{vp}[i] = 1$. It should be noted the the message from node c is not included in the computation. This is known as the extrinsic information principle [9] and is the basis for all belief propagation techniques.

Check node messages are more complex to compute, in [9] it is stated that $V_{cv}[k]$, $k = 0$ or 1, is computed by multiplying all input message values, while observing the extrinsic information principle, and summing the results. Summations where the indices of the product terms add up to k using modulo-2 addition are assigned to $V_{cv}[k]$. This can be summarized as:

$$V_{cv}[k] = \sum_{\sum i_j = k \bmod 2} \prod_{j=1, j\neq v}^{d_c} U_{jc}[i_j] \tag{2.3}$$

As mentioned in [10], this is a convolution of the incoming messages and can be written as:

$$V_{cv} = \circledast_{i=1, i\neq v}^{d_c} U_{ic} \tag{2.4}$$

where $\circledast$ is the convolution operator.

A method commonly used to simplify SPA decoding without affecting performance is to use log-likelihood ratios (LLR) as messages [9]. To compute the LLR $\hat{X}$ of a likelihood vector X, one uses $\hat{x} = \log \frac{P[x=1|y]}{P[x=0|y]}$. This has the effect of changing multiplications in (2.2) to additions, so that it becomes:

$$\hat{U}_{vc} = \hat{L} + \sum_{i=1, i\neq c}^{d_v} \hat{V}_{iv} \tag{2.5}$$

and converting (2.3) into:

$$\hat{V}_{cv} = 2 \tanh^{-1}\left(\prod_{i=1, i\neq v}^{d_c} \tanh\left(\frac{\hat{U}_{ic}}{2}\right)\right) \tag{2.6}$$

2.2 LDPC Codes over GF(q)

Like binary codes, non-binary LDPC codes are defined by a sparse parity check matrix H. However, the elements of H are elements of a Galois field GF(q) and the arithmetic is GF(q) arithmetic. Most works in the literature restrict q such that $q = 2^p$ since this greatly simplifies algorithms and their implementation.

Gallager [6] suggested that non-binary LDPC codes could be constructed using modulo-q arithmetic. However, it was Davey et al [2] who first studied these codes in detail and used operations over GF(q) for decoding.

Results from literature show that non-binary LDPC codes can outperform binary LDPC codes of equivalent length and RS codes by up to multiple dBs even when the errors introduced by the channel are correlated [3] and [11].

Many properties such as row and column weights and decodability over graphs hold true for non-binary codes. In section 2.2.1, differences between binary and GF(q) SPA decoding are presented. Various simplifications to GF(q) decoding algorithms are discussed in sections 2.2.1 - 2.2.5.

2.2.1 SPA Decoding

While non-binary codes are also decoded over Tanner graphs, the decoding algorithm is not a direct generalization of the binary case because the elements of H are non-binary. As a result a check node constraint is:

$$\sum_{k=1}^{d_c} h_k \beta_k = 0 \tag{2.7}$$

where h_k is the element of H with indices corresponding to the check and variable nodes of communicating with each other and β_k is a GF(q) symbol corresponding

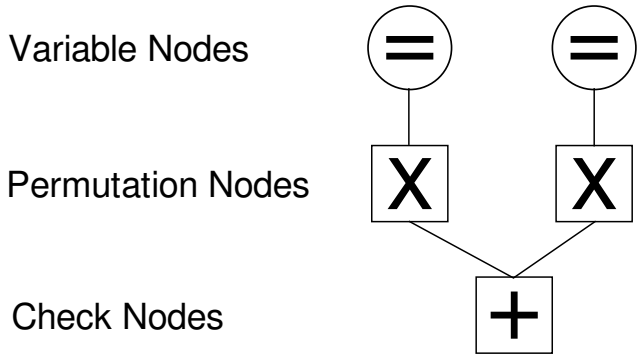


Figure 2–2: A portion of a GF(q) Tanner graph showing the three node types.

to a check node input message . This is different from the binary case where the check constraint is $\sum_{k=1}^{d_c} \beta_k = 0$. To accommodate this change, Davey et al [2] assign values from H as labels to the graph edges connecting the variable and check nodes and integrate the multiplication into the check node functionality. Declercq et al [10] present a different approach: they introduce a third node type called the permutation node which connects variable and check nodes and performs multiplication as shown in Figure 2–2; therefore reverting the check node constraint to

$$\sum_{k=1}^{d_c} \beta_k = 0 \tag{2.8}$$

While the two approaches are functionally equivalent; the one in [10] results in simpler equations and implementation since all check nodes of the same degree are identical. Therefore, it is the one utilized in out work.

Due to the non-binary nature of the code, the messages transmitted between the node types are q-element PMF vectors indexed using GF(q) symbols. For example $U_{vp}[\beta]$ is the probability of symbol β.

Like the binary SPA, The first step in GF(q) SPA is computing the channel likelihood vector using $L[\beta] = \prod_{k=1}^{p} l(i_k = \beta_k)$, where $l(i_k = \beta_k)$ is the probability of bit k in the received symbol being equal to bit k in the polynomial representation of GF(2^p) symbol β, and is computed just like in the binary case.

The variable node update message equation remains the same as equation (2.2) from the binary case. However the messages arrive from permutation, not check, nodes. The size of vectors also changes, and the equation shown in [10] is:

$$U_{vp} = L \times \prod_{i=1, i \neq p}^{d_v} V_{iv} \tag{2.9}$$

where $\times$ is the term-by-term product of vectors. Normalization is still needed in the non-binary case so that $\sum_{i=1}^{q} U_{vp}[i] = 1$.

The permutation nodes implement multiplication by a GF(q) element when passing messages from the variable to check nodes, and multiplication by its inverse in the other direction. Since GF(q) are cyclic fields, the multiplication and division can be performed by cyclic shifts of all values in a message except those indexed by 0 as in [10]. The equation corresponding to passing messages from variable to check nodes can be written as:

$$U_{pc}[i] = U_{vp}[i * h_p] \tag{2.10}$$

where h_p is the H matrix element corresponding to the permutation node, and $*$ is GF(q) multiplication.

For messages passed in the other direction the following is used:

$$U_{pc}[i] = U_{vp}[i * h_p^{-1}] \tag{2.11}$$

where h_p^{-1} is the GF(q) inverse of h_p.

Since the parity check constraint does not include multiplication by elements of H anymore, the check node update equation takes a similar form to the binary SPA equation (2.3):

$$\begin{aligned} V_{cv}[k] &= \sum_{\sum i_j + k = 0} \prod_{j=1, j\neq v}^{d_c} U_{jc}[i_j] \quad \text{so that:} \\ V_{cv} &= \circledast_{i=1, i\neq v}^{d_c} U_{ic} \end{aligned} \tag{2.12}$$

where the addition in the constraint $\sum i_j + k = 0$ is over GF(q).

The complexity of computing the check node's output messages using equation (2.12) is given by [10] as: $O(d_c q^{d_c - 1})$, rendering a direct implementation of the SPA impractical. Davey et al. [2] suggest using partial summations to evaluate the equation as a method to reduce complexity. However it remain impractical as Declercq *et al.* [10] show the complexity to be $O(d_c q^2)$.

2.2.2 FFT-SPA and Log-FFT-SPA Decoding

Since the complexity of equation (2.12) increases exponentially with both field order q and check node degree d_c, other methods for computing the check node output must be utilized. The authors of works [3, 12, 13] have all proposed performing the computation in the frequency domain. This can be accomplished using the Fourier transform. If the codes are over GF(2^p), the Fourier transform is a Hadamard transform [14] which can be computed using $W = \mathcal{F}(U) = UH_m$, where H_m is the Hadamard matrix which can be defined recursively as:

$$H_m = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{m-1} & H_{m-1} \\ H_{m-1} & -H_{m-1} \end{bmatrix} \tag{2.13}$$

with $H_1 = 1$.

As shown in [10], using this method equation (2.12) becomes:

$$V_{cp} = \mathcal{F}\left(\prod_{i=1, i\neq p}^{d_c} \mathcal{F}(U_{ic})\right) \tag{2.14}$$

This reduces the complexity of computing a check node's messages to $O(d_c pq)$ when using the fast Hadamard transform as indicated by [10].

It should be noted that the normalization factor $\frac{1}{\sqrt{2}}$ in the Hadamard matrix is not needed since the variable node performs normalization on its output messages. This was confirmed by our simulation results.

The performance of this decoding method is identical to that of the sum-product algorithm.

Song *et al.* [3] proposed using logarithms to convert multiplications into additions in the FFT-SPA decoder. They perform all operations on the sign and magnitude of the message elements independently. This enabled them to implement a decoder using addition and subtraction only without any multipliers or dividers. Without any approximations, this algorithm performs identically to SPA.

2.2.3 LLR-SPA Decoding

As in binary SPA, LLR messages can be used instead of PMFs to convert multiplications into additions. This was done by Wymeersch *et al.* in [15]. An LLR message can be generated from a PMF one using:

$$\hat{U}[\beta] = \log \frac{U[\beta]}{U[0]}, \quad \beta \in \mathrm{GF}(2^p)$$

Using LLR messages, the initial likelihood vector $\hat{L}$ for a received symbol x is given by:

$$\begin{aligned} \hat{L}[\beta] &= \log \frac{l(x_1 = \beta_1)l(x_2 = \beta_2)...l(x_p = \beta_p)}{l(x_1 = 0)l(x_2 = 0)...l(x_p = 0)} \\ &= \sum_{k=1,\beta_k \neq 0}^{p} \log \frac{l(x_k = 1)}{l(x_k = 0)} \end{aligned} \tag{2.15}$$

As noted in [15], if β contains zeros in its binary representation, as all but one GF(q) element do, some terms cancel in the equation. Thus, the summation does not include p terms.

The variable node update message uses additions in place of multiplications; so equation (2.9) becomes:

$$\hat{U}_{at} = \hat{L} + \sum_{p=1,p\neq t}^{d_v} \hat{V}_{pa} \tag{2.16}$$

No normalization is needed since LLR messages are used.

The permutation node is still implemented using cyclic shifts.

The check node update messages are computed using partial sums. As such, its complexity is still $O(d_c q^2)$.

The performance of this decoding method is also identical to that of the sum-product algorithm when no approximation are performed.

2.2.4 Extended Min-Sum (EMS) Decoding

The extended min-sum algorithm (EMS), presented in [10], reduces the complexity of computing equation (2.12) by using LLR messages and reducing the number of input messages considered in the computation. Only the largest n_m values of each $\hat{U}_{pc}$ are considered, and these values are denoted $\hat{u}_{pc}^{(k_c)}$, $k_c = 1, 2, ..., n_m$ and can be

found using:

$$\hat{u}_{pc}^{(k_c)} = \hat{U}_{pc}[\alpha_c^{(k_c)}] \tag{2.17}$$

where $\alpha_c^{(k_c)}$ is the field element to which the likelihood $\hat{u}_{pc}^{(k_c)}$ corresponds.

The n_m most likely symbols of each of the $d_c - 1$ check node inputs involved in computing an output message are used to form $d_c - 1$ sets. The Cartesian product of these sets is the configuration set $\text{Conf}(n_m)$ which is formally defined in [10] as:

$$\begin{aligned}\text{Conf}(n_m) = \{\boldsymbol{\alpha}_k = [\alpha_1^{(k_1)}(x), ..., \alpha_{d_c-1}^{(k_{d_c-1})}(x)]^T : \\ \forall \mathbf{k} = [k_1, ..., k_{d_c-1}]^T \in \{1, ..., n_m\}^{d_c-1}\}\end{aligned} \tag{2.18}$$

The number of elements in $\text{Conf}(n_m)$ is $|\text{Conf}(n_m)| = n_m^{d_c-1}$ since only the largest n_m likelihood values are considered from each inbound message. Conf(1) contains only one configuration, representing the vector of symbols with largest likelihood values at each input of the node, and is called the order-0 configuration.

To further reduce the number of configuration considered when computing an output message, Declercq et al. [10] restrict the computations to the following subset of $\text{Conf}(n_m)$:

$$\begin{aligned}\text{Conf}(n_m, n_c) = \text{Conf}(n_m)^{(0)} \cup \text{Conf}(n_m)^{(1)} \cup ... \\ \cup \text{Conf}(n_m)^{(n_c)}\end{aligned} \tag{2.19}$$

where $n_c \leq d_c - 1$ and $\text{Conf}(n_m)^{(k)}$ is the subset of configurations that differ from the order-0 configuration by k elements. The cardinality of $\text{Conf}(n_m, n_c)$ is given in

[10] as:

$$\begin{aligned} |\mathrm{Conf}(n_m, n_c)| &= \sum_{k=0}^{n_c} \begin{pmatrix} d_c - 1 \\ k \end{pmatrix} (n_m - 1)^k \\ &\approx \begin{pmatrix} d_c - 1 \\ n_c \end{pmatrix} n_m^{n_c} \end{aligned} \tag{2.20}$$

The configurations which satisfy the check node equation (2.8) are denoted $\mathrm{Conf}_{i_{d_c}}(n_m, n_c)$ and are defined by:

$$\mathrm{Conf}_{i_{d_c}}(n_m, n_c) = \{\boldsymbol{\alpha_k} \in \mathrm{Conf}(n_m, n_c) : i_{d_c} + \sum_{c=1}^{d_c-1} \alpha_c^{(k_c)} = 0\}$$

where $i_{d_c} \in \mathrm{GF}(q)$.

A measure of reliability is also introduced in [10] and is computed using the equation:

$$L(\boldsymbol{\alpha_k}) = \sum_{c=1}^{d_c-1} \hat{u}_c^{(k_c)}$$

In the EMS algorithm, the variable and permutation node equations remain the same as in LLR-SPA (Section 2.2.3). The check node update message becomes:

$$\hat{V}_{d_c p}[i_{d_{c_1}}, ..., i_{d_{c_p}}] = \max_{\boldsymbol{\alpha_k} \in S_{i_{d_c}}} \{L(\boldsymbol{\alpha_k})\} \tag{2.21}$$

where the set $S_{i_{d_c}}$ is defined as

$$S_{i_{d_c}} = \mathrm{Conf}_{i_{d_c}}(q, 1) \cup \mathrm{Conf}_{i_{d_c}}(n_m, n_c)$$

The set of configurations $\mathrm{Conf}_{i_{d_c}}(q, 1)$ is needed in the above equation to guarantee that $S_{i_{d_c}}$ is never empty. To avoid numerically saturating the LLR values and maintain consistency with the LLR definition, the result from equation (2.21) needs to

be post-processed as:

$$\hat{V}_{cp}[\beta] = \hat{V}_{cp}[\beta] - \hat{V}_{cp}[0], \quad \beta \in \mathrm{GF}(q)$$

Since the EMS discards some messages when computing the check node messages, it results in a performance loss. To mitigate the degradation, Declercq et al. [10] either use offset factors, or apply scaling to check node messages. The results in [10] show that with correction, EMS performs within 0.1 dB of the SPA when n_m and n_c are sufficiently large. It is also noted that the performance of EMS with a scaling factor degrades with increasing SNR values, while the offset correction does not show such a trend.

The complexity of computing a check node's output messages with the EMS algorithm with the corrections applied is given in [10] as $O(d_c q \log q) = O(d_c p 2^p)$ when $q = 2^p$, which is lower any of the previously mentioned algorithms.

An similar approach to EMS is presented in [16]. The difference is that [16] uses dynamic programming instead of configuration sets when computing the check node messages. This reduces the complexity of computations and does not require corrections the message values. [16].

2.2.5 Reduced Complexity EMS Decoding

Voicila et al. [17] describe a reduced complexity version of the EMS decoder. The original EMS algorithm used only n_m values when computing check node messages; however, it stored and transmitted all q likelihood values for any message. The new algorithm in [17] uses and stores only n_m messages. The messages in this algorithm are passed as vectors sorted in order of decreasing value of likelihood and indexed from 0. Since the messages are sorted, the mapping of the index of

messages in the likelihood vector to GF(q) is not constant. Therefore, a mapping vector is needed. For example, let $\mathbf{A}$ be an LLR message. $\mathbf{B}$ is a vector composed of the largest n_m values of $\mathbf{A}$ sorted in decreasing order. β_B is the mapping vector from indices of $\mathbf{B}$ to GF(q) elements. The authors in [17] also append a value γ_A to B that represents an average likelihood of the discarded elements of $\mathbf{A}$. Thus, $\mathbf{B} = [\mathbf{B}[0], \mathbf{B}[1], ..., \mathbf{B}[n_m - 1], \gamma_A, ..., \gamma_A]$ and is of length q. However, since the last $q - n_m$ elements of $\mathbf{B}$ all have the same value γ_A, that value needs only be stored once. Also since $\mathbf{B}$ is sorted in decreasing order, $\gamma_A \leq \mathbf{B}[n_m - 1]$.

Assuming the LLR values in $\mathbf{B}$ correspond to a normalized PMF, the value of γ_A is given in [17] as:

$$\gamma_A = \log \sum_{i=0, A[i] \notin B}^{q-1} e^{A[i]} - \log(q - n_m) \tag{2.22}$$

The previous equation can be approximated using $\max^*(x_1, x_2) = \log(e^{x_1} + e^{x_2}) \approx \max(x_1, x_2)$. Applying the approximation results in equation (2.22) becoming:

$$\gamma_A \approx B[n_m] - \log(q - n_m) \tag{2.23}$$

The approximation used is known to over-estimate the LLR values. As a result, an offset factor is used in [17]. Since, $\log(q - n_m)$ is also constant for a particular decoder, it can be combined with the offset factor. Equation 2.23 becomes:

$$\gamma_A \approx B[n_m] - \text{offset} \tag{2.24}$$

The offset value is computed in [17] by maximizing the threshold of the LDPC code, which in turn is computed via the density evolution algorithm.

The steps of the reduced complexity EMS algorithm as listed in [17] are:

1. Initialize the decoder by selecting the n_m largest values from the initial likelihood messages $\hat{L}$.
2. Compute the variable node messages.
3. Perform the permutation node operations by modifying the mapping vector β. This accomplished by a GF(q) multiplication.
4. Compute the check node messages.
5. Perform the inverse permutation operations.

Steps 2 and 4 are performed via a recursive implementation where each node is decomposed into a number of elementary steps. Each elementary step has two inputs and one output. The overall node output message is computed from a intermediate message at the output of an elementary step and an input message. The node construction as described in [17] is discussed below.

Variable Node: An elementary step in a variable node has inputs **I** and **V**, and output **U**. Where **I**, **V**, and **U** have length n_m and mapping vectors β_V, β_I, and β_U. An internal vector T of length $2n_m$ is computed as follows:

$$T[k] = V[k] + Y, \quad T[n_m + k] = \gamma_V + I[k], \quad k \in \{0, ..., n_m - 1\}$$

where

$$Y = \begin{cases} I[l] & \text{if } \beta_I[l] = \beta_V[k] \quad k, l \in \{0, ..., n_m - 1\} \\ \gamma_I & \text{if } \beta_I[l] \notin \beta_V \end{cases}$$

The output message U is the largest n_m values of T.

Check Node: An elementary step in a check node has inputs **U** and **I**, and output **V**. These vectors are defined in a manner similar to that in the variable node. If $S(\beta_V[i])$ is defined as the set of GF(q) elements such that $\beta_V[i] \oplus \beta_U[j] \oplus \beta_I[p] = 0$,

then the output $\mathbf{V}$ is:

$$V[i] = \max_{S(\beta_V[i])} (U[j] + I[p]) \tag{2.25}$$

Since $\mathbf{U}$ and $\mathbf{I}$ are sorted in decreasing order, the authors in [17] propose a method that exploits the ordering to reduce the number of computations required to find the largest n_m values of $(U[j] + I[p])$. They propose constructing a virtual matrix M whose elements are $M[j, p] = U[j] + I[p]$. The largest n_m values in M are all located in the upper anti-diagonal. The proposed algorithm steps are:

1. Initialization: The elements of the first column of M are shifted into a sorter.
2. The largest value is computed in the sorter and is sent as an output.
3. If the associated likelihood value of the GF(q) element satisfying the check constraint already exists in the output vector, no action is taken. Otherwise, the value is written to $\mathbf{V}$.
4. The right neighbor of the M element from the previous step is introduced into the sorter.
5. Steps 2-4 are repeated K_{max} times or until all n_m values in $\mathbf{V}$ correspond to unique GF(q) elements.

It was determined in [17] that $K_{max} = 2n_m$ yields very good results. In the cases where K_{max} iterations do not result in n_m values in $\mathbf{V}$, γ_V is used to pad the output vector.

The results of a reduced complexity EMS decoder in [17] show that it performs within a faction of a dB for sufficiently larger n_m values. In these results we also note that the algorithm performs worse on $GF(256)$ than on $GF(64)$ for a the same value of n_m.

2.3 Stochastic Decoding

Stochastic computation was introduced as method for designing low-precision, lower-cost digital circuits [18]. Data is represented by a streams of random symbols whose statistics represent the message being transmitted. Its main advantage is that the symbols can be manipulated directly by simple circuitry to the change the message the stream represents.

2.3.1 Binary LDPC Stochastic Decoding

Sharifi Tehrani et al. [19] introduced a low-complexity decoder based on stochastic computation [18, 20] for decoding binary LDPC codes. The stochastic streams passed in the decoder are Bernoulli sequences, where the number of occurrences of a symbol (0 or 1) in the stream divided by the length of the stream corresponds to the probability value of that symbol. For example a stream 0001001011 indicates that $p_0 = 7/10$ and $p_1 = 3/10$.

In [19], the authors only track p_1 since $p_0 = 1 - p_1$, and define the node output using only that probability value; so when p_c is used, it refers to $P[c = 1]$.

Since the stochastic messages are streams, an index t is used to refer to the symbol's location in a stream, e.g. $\overline{U}(t)$ is the symbol at location t in stream $\overline{U}$.

Like the SPA decoder, a binary stochastic decoder implements a Tanner with variable and check nodes. The stochastic variable node implements the SPA variable node message by directly manipulating the input stochastic streams. In addition to the streams from check nodes, the variable node has an input stream which is generated using channel likelihood values as a PMF. The variable node update rule

is given in [19] as:

$$\overline{U}_{vc}(t) = \begin{cases} 0 & \text{if } \overline{V}_{iv} = 0, \forall i : i \neq c \\ 1 & \text{if } \overline{V}_{iv} = 1, \forall i : i \neq c \\ \overline{U}_{vp}(t-1) & \text{otherwise} \end{cases} \tag{2.26}$$

The output stream has a PMF that is equal to the SPA variable node's output message.

A stochastic check node's output message is computed using:

$$\overline{V}_{cv}(t) = \oplus_{i=1, i\neq v}^{d_c} \overline{U}_{ic} \tag{2.27}$$

where $\oplus$ is the XOR operator (GF(2) addition). The PMF of the resulting stream for a degree-3 stochastic check node output is shown in [19] to be $p_{cv} = (1-p_{1c})p_{2c} + p_{1c}(1-p_{2c})$ which is the same for a similar SPA node.

The authors in [19] note that if a stochastic decoder is implemented directly using the previous two rules, it would have severely degraded performance. They propose scaling the channel likelihood values and randomly shuffling the variable node output streams as modifications for the decoder to function properly. With these enhancements, the performance results in [19] show that the binary stochastic decoder can match an SPA based one. These modifications are examined in more detail in Section 3.2.

2.3.2 Trellis-Based Stochastic Decoding

Winstead *et al.* [21] presented another stochastic algorithm for decoding binary codes that uses streams of integers as decoder messages.

The integers incoming into a variable node are checked against a constraint and if they satisfy it, the node output is updated; otherwise, the old output value is retained. This is similar to the binary stochastic update rule, but generalized for integers and is summarized as:

$$\overline{U}_{vp}(t) = \begin{cases} a & \text{if } \overline{V}_{iv} = a, \forall i : i \neq p \\ \overline{U}_{vp}(t-1) & \text{otherwise} \end{cases} \tag{2.28}$$

The check node input messages are used to encode the probabilities of the states in a trellis according to a particular mapping. This trellis performs the convolution in Equation (2.12).

The algorithm in [21] is used to decode a (16,11)-bit Hamming code, and a (256,121) turbo block code.

Chapter 3

An Algorithm for Stochastic Decoding of LDPC over GF(q)

This chapter describes in detail the stochastic algorithm for decoding LDPC codes over GF(q). First the notation used and the nature of decoder messages are presented. In Sections 3.1.1 - 3.1.3, the node equations are developed and compared with those of the SPA. Two modifications critical to decoder functionality are presented in Section 3.2. The performance of the algorithm is presented in Section 3.4.

In Section 3.5, we show that the binary stochastic decoding algorithm of [22] is a special case of our algorithm.

3.1 Node Equations

While a message in SPA for LDPC codes over GF(q) is a vector containing the probabilities of each of the q possible symbols, in stochastic decoding messages are represented using streams of GF(q) symbols. As in the binary stochastic case, the number of occurrences of a symbol in a stream divided by the total number of symbols in the stream corresponds to the probability value of that symbol. The PMF of a symbol β as conveyed by a stream c of length m is calculated by the following

equation:

$$\text{PMF}[\beta] = \frac{1}{m}\sum_{k=1}^{m} I(c[k] = \beta)$$

where $I(c[k] = \beta) = 1$ when $c[k] = \beta$ and 0 otherwise.

A notation similar to the SPA is used when denoting the stochastic messages, the difference being that messages are serial stochastic streams instead of vectors; thus, an index t is used to denote the location of a symbol within a stream and the stream label is over-lined, e.g. $\overline{U}_{vp}(t)$.

3.1.1 Variable Node

A stochastic variable node of degree d_v takes as inputs d_v stochastic streams from permutation nodes in addition to one generated based on channel likelihood values associated with the node which we call the channel stream. As was proposed in [21], the output of the node is updated if the inputs satisfy a constraint; otherwise, the output remains unchanged from the previous iteration. To implement the constraint at time t, we copy the input symbol to the output symbol on a certain edge if the input symbols of all other incoming edges, including the channel stream, are equal at time t.

For a variable node with output $\overline{U}_{vp}$ and inputs $\overline{V}_{iv}$, the update rule is:

$$\overline{U}_{vp}(t) = \begin{cases} a & \text{if } \overline{V}_{iv} = a, \forall i : i \neq p \\ \overline{U}_{vp}(t-1) & \text{otherwise} \end{cases} \tag{3.1}$$

If we assume that the inputs to a variable node are independent and use Equation (3.1), PMF of an output stream is:

$$\begin{aligned} P[\overline{U}_{vp}(t) = c] &= \prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = c] \\ +(1 - \sum_{a\in \mathrm{GF}(q)} &\prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = a]) P[\overline{U}_{vp}(t-1) = c] \end{aligned} \tag{3.2}$$

If the stochastic streams are assumed to be stationary as in [21], then $P[\overline{U}_{vp}(t) = c] = P[\overline{U}_{vp}(t-1) = c]$ and the PMF of $\overline{U}_{vp}(t)$ becomes:

$$P[\overline{U}_{vp}(t) = c] = \frac{\prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = c]}{\sum_{a\in \mathrm{GF}(q)} \prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = a]}. \tag{3.3}$$

Equation (3.3) is identical to the normalized output of the SPA variable node in Equation (2.9). Therefore we conclude that Equation (3.1) is a valid update rule for the stochastic variable node.

3.1.2 Permutation Node

The function of the permutation node is decoupling multiplication by elements of H from the check node constraint. In the sum-product algorithm, this is achieved by a cyclic shift of the message vector elements as in Section 2.2.1. Here, we demonstrate that multiplying the stochastic stream from a variable to a check node by an element of H accomplishes the same result. Assuming a permutation node p corresponding to H element $h = \alpha^i$, the permutation node output message in an SPA decoder is

defined such that each element in the message vector is given by:

$$U_{pc}[a] = U_{vp}[a.\alpha^i], \quad \forall\, a \in \mathrm{GF}(q).$$

In a stochastic decoder, when the permutation node multiplies all elements of the input by h, the output PMF becomes:

$$P[\overline{U}_{pc}(t) = a] = P[\overline{U}_{vp}(t) = a.\alpha^i]$$

The SPA and stochastic output PMFs are identical and since the multiplicative group of GF(q) is cyclic and multiplication is closed on GF(q), the stochastic permutation node operation is equivalent to that of the SPA algorithm.

Similarly, it can be shown that for messages passed from check to variable nodes, the inverse permutation node operation is multiplication by h^{-1}.

It should be noted that $h \neq 0$, since a value of 0 in H signifies the lack of a connection between a variable and a check node. Therefore, there are no permutation nodes with a multiplier $h = 0$.

3.1.3 Check Node

When deriving the stochastic update message for a check node, we start with a degree-three node and generalize the resulting equation to a check node of any degree.

Let U_{1c} and U_{2c} be the node inputs, which are assumed to be independent, and V_{cp} be its output. From Equation (2.12), the output of such a node when using the SPA is given as:

$$P[V_{cp} = z|U_{1c}, U_{2c}] = \sum_{x \oplus y = z} P[U_{1c} = x]P[U_{2c} = y], \tag{3.4}$$

where $\oplus$ is GF(q) addition.

In the stochastic node, we define the output as the GF(q) addition of inputs, i.e $\overline{V}_{cp}(t) = \overline{U}_{1c}(t) \oplus \overline{U}_{2c}(t)$. The PMF of the output is computed as:

$$\begin{aligned} P[\overline{V}_{cp}(t) = z] &= P[\overline{U}_{1c}(t) \oplus \overline{U}_{2c}(t) = z] \\ &= \sum_{x \oplus y = z} P[\overline{U}_{1c}(t) = x] P[\overline{U}_{2c}(t) = y]. \end{aligned} \tag{3.5}$$

The PMFs (3.4) and (3.5) are identical; therefore it is concluded that GF(q) addition is a valid update message for a degree-3 stochastic check node.

Since the output of a check node can be computed recursively [10], the previous conclusion can be generalized to a check node of any degree, and the output messages for these nodes are given as:

$$\overline{V}_{cp}(t) = \sum_{i=1, i \neq p}^{d_c} \overline{U}_{ic}(t), \tag{3.6}$$

where the summation is over GF(q).

As can be seen from Equation (3.6), the check node update message complexity is $O(d_c - 1)$ which is linear in d_c and indepent of the field order q. In Section 4.4, we show that the complexity of computing all of a check node's outgoing messages remains linear in d_c and is given as $O(2d_c - 2)$.

3.2 Noise-Dependent Scaling and Edge Memories

In stochastic decoding, switching activity in message streams can become very low resulting in poor bit-error-rate performance. This phenomenon is called latch-up and is caused by cycles in the graph that cause the stochastic streams to become

correlated invalidating the independent stream assumption used to derive Equations (3.1) and (3.6). Two solutions were proposed in [19]: noise-dependent scaling and edge memories. Both of these methods are used to improve the performance of the GF(q) decoder.

Noise-dependent scaling increases switching activity by scaling the channel likelihood values. For example, when transmitting data using BPSK modulation over an AWGN channel the scaled likelihood of each received bit $l'(i)$ is calculated by:

$$l'(i) = [l(i)]^{\frac{2\alpha\sigma_n^2}{Y}},$$

where $l(i)$ is the unscaled bit likelihood, σ_n^2 is the noise variance, and the ratio $\frac{\alpha}{Y}$ is determined offline to yield the best performance in the SNR range of interest. Accordingly the equation for computing the channel likelihood values becomes:

$$L[\beta] = \prod_{k=1}^{p} [l(\beta_k)]^{\frac{2\alpha\sigma_n^2}{Y}}. \tag{3.7}$$

Edge memories (EM), in their simplest form, are finite depth buffers inserted between variable nodes and permutation nodes and randomly reorder symbols in the output streams of variable nodes; thus, they break correlation between streams without affecting the overall stream statistics. The EM contents are updated with the variable node output when the node update condition is satisfied, and remain intact otherwise. The output of the EM is that of the variable node in the first case, or a randomly selected symbols from its contents in the second. Due to the memory's finite length, older symbols are discarded when new ones are added.

Figure 3–1 demonstrates the message passing mechanism and the location of edge memories within a stochastic decoder.

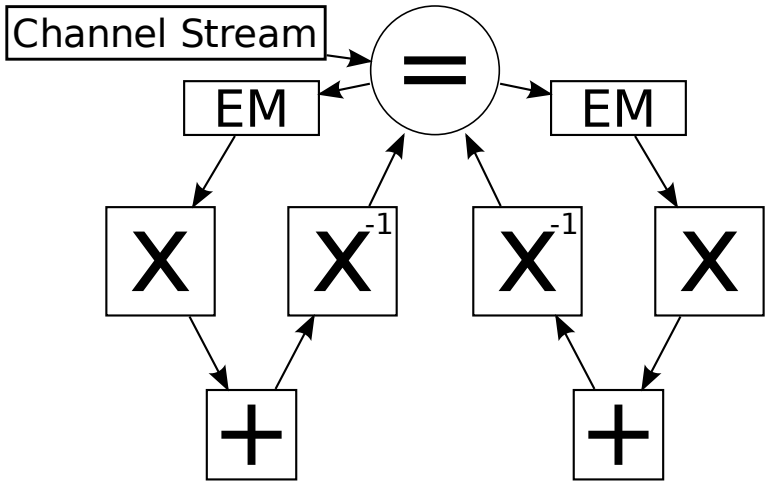


Figure 3–1: Message propagation in the stochastic decoder.

3.3 Algorithm Description

The first step in decoding a received vector is to compute the channel likelihood values based on received soft information and channel model. Furthermore, NDS is applied to these values resulting in a scaled version which is then normalized. Afterwards, each edge memory is initialized such that the distribution of its contents has as its PMF the scaled channel likelihood values corresponding to the variable node the edge memory is connected to. The iterative decoding process then begins and the following steps describe the decoder functions performed during reach decoding cycle.

1. Variable node messages are computed using Equation (3.1). Edge memory contents are updated where appropriate and messages are sent from edge memories to permutation nodes.
2. Permutation nodes multiply, over GF(q), the incoming messages by the elements of H and send the results to check nodes.

3. Check node messages are computed using Equation (3.6) and sent to permutation nodes.
4. Permutation nodes multiply the incoming messages by the inverse of the elements of H and the send the results to the variable nodes.
5. Each variable node contains q counters $C[a]$ corresponding to GF(q) elements that are used to track the variable node belief. A counter corresponding to a particular symbol is incremented when all of the variable node inputs, including the channel stream, are equal to that symbol. The variable node belief is defined as $\arg\max C[a]$.

The message streams are processed on a symbol-by-symbol basis, one symbol each cycle (steps 1-5), until the algorithm converges or a maximum number of decoding cycles is reached. The algorithm is said to converge if the variable node beliefs satisfy the check constraints, i.e. the beliefs are used to perform syndrome checking. The last step of the decoding algorithm is to present the variable node beliefs as the decoder output.

3.4 Performance

Figure 3–2 demonstrates the performance of the stochastic decoder compared to that of an SPA decoder when decoding a (256,128)-symbol, rate 1/2 LDPC code over GF(16) [23], when using an AWGN channel, BPSK, and random source data. The SPA decoder has a maximum of 1000 iterations, while the stochastic decoder's maximum is 10^6 decoding cycles (DC). The performance of the two decoders is very similar and the two decoders perform identically for higher SNR values. The change in the slope of the error rate graph was also observed in [23].

SNR (dB)	2.0	2.5	3.0	3.5	4.0
DC_{avg} ($DC_{\text{max}} = 10^6$)	22599	8888	4243	2329	1433
DC_{avg} ($DC_{\text{max}} = 10^5$)	17958	8511	4209	2326	1433

Table 3–1: Average number of decoding cycles.

The maximum number of decoding cycles is much greater than the average number of decoding cycles as shown in Table 3–1, with DC_{avg} determining the decoder throughput which is given in information bits per clock cycle. In Figure 3–2 we can see that, at higher SNRs, DC_{max} can be reduced with a small performance loss.

It should be noted that the number of iterations in the SPA decoder and decoding cycles in the stochastic decoder are not directly comparable. SPA iterations involve complex operations, for example, the node operations in EMS [24] involve sorting and iterating over incoming message elements; thus, requiring many clock cycles. In a stochastic decoder, a decoding cycle is very simple and can be completed within a single clock cycle. Also, due to the nature of stochastic computation, the proposed implementation lends itself to pipelining (due to the random order of the messages, the feedback loop in the graph is broken allowing pipelining [22]); thus, enabling clock rates faster than those possible with the SPA. A detailed complexity analysis is presented in Section 4.6.

3.5 Binary Stochastic Decoding as a Special Case

In this section we prove that GF(q) stochastic decoding of codes with $q = 2$ has the same output messages, stream statistics, and results as the binary stochastic

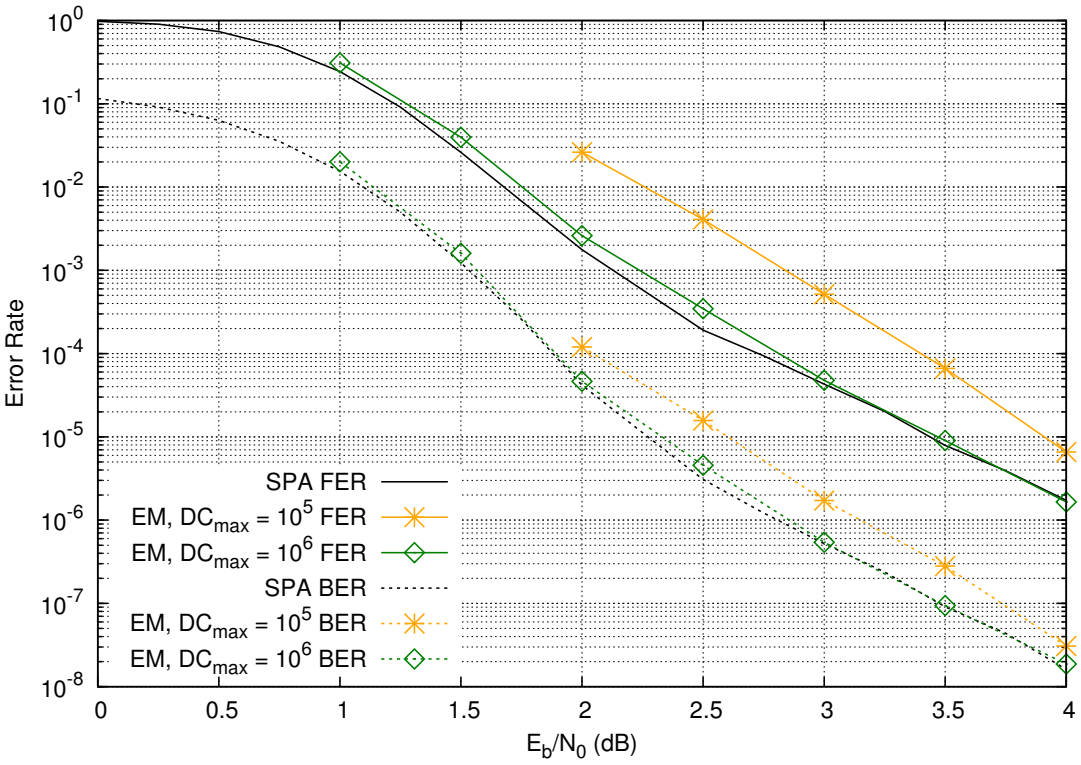


Figure 3–2: FER and BER for a (256,128)-symbol (2,4)-regular LDPC code over GF(16). EM length = 50, $\frac{\alpha}{Y} = 0.5$.

algorithm of [22]. We refer to the proposed algorithm as GF(2) stochastic decoding and to that of [22] as binary stochastic decoding.

When operating over GF(2), we only have two symbols: 0 and 1. Therefore, we only need to track the probability of one of them because the probabilities are related by: $P[x = 0] = 1 - P[x = 1]$. In this section we choose to track $P[x = 1]$.

A GF(2) variable node has the following update rule derived from Equation (3.1):

$$\overline{U}_{vp}(t) = \begin{cases} 0 & \text{if } \overline{V}_{iv} = 0, \forall i : i \neq p \\ 1 & \text{if } \overline{V}_{iv} = 1, \forall i : i \neq p \\ \overline{U}_{vp}(t-1) & \text{otherwise} \end{cases} \tag{3.8}$$

This is the same update rule proposed in [22].

Furthermore, starting with the GF(q) variable node output message PMF in Equation (3.3), we obtain:

$$\begin{aligned} P[\overline{U}_{vp}(t) = c] &= \frac{\prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = c]}{\sum_{a \in \mathrm{GF}(2)} \prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = a]} \\ &= \frac{\prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = c]}{\prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = 0] + \prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = 1]} \\ &= \frac{\prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = c]}{\prod_{i=1, i\neq p}^{d_v} (1 - P[\overline{V}_{iv}(t) = 1]) + \prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = 1]} \end{aligned} \tag{3.9}$$

Since we only track $P[x = 1]$, from Equation (3.9) we have:

$$P[\overline{U}_{vp}(t) = 1] = \frac{\prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = 1]}{\prod_{i=1, i\neq p}^{d_v} (1 - P[\overline{V}_{iv}(t) = 1]) + \prod_{i=1, i\neq p}^{d_v} P[\overline{V}_{iv}(t) = 1]} \tag{3.10}$$

Which matches exactly the output message PMF of binary stochastic decoding.

In binary LDPC codes, all non-zero H elements are equal to 1; thus, the permutation and inverse permutation node multipliers are all equal to 1. Since multiplication by 1 does not affect the stochastic streams, the permutation nodes can be removed when decoding binary codes.

From Equation (3.6), we know that the update rule for a stochastic GF(2) check node is:

$$\overline{V}_{cp}(t) = \sum_{i=1, i\neq p}^{d_c} \overline{U}_{ic}(t), \tag{3.11}$$

where the addition is performed over GF(2). GF(2) addition is an XOR operation; thus making Equation (3.11) and the binary stochastic check node update rule identical. The probability of a degree-3 GF(2) check node output symbol being equal to 1 is calculated from the PMF (3.5) as:

$$P[\overline{V}_{cp}(t) = 1] = P[\overline{U}_{1c}(t) \oplus \overline{U}_{2c}(t) = 1] \tag{3.12}$$

$$= P[\overline{U}_{1c}(t) = 1](1 - P[\overline{U}_{2c}(t) = 1]) + \ (1 - P[\overline{U}_{1c}(t) = 1])P[\overline{U}_{2c}(t) = 1].$$

which is the same as that of a degree-3 stochastic binary check node.

Based on this comparison, we conclude that the binary stochastic decoding of [22] is a special case of GF(q) decoding when operating over GF(2). It should be noted, however, that the architecture of the binary stochastic decoder in [22] differs slightly from that of the GF(2) decoder due to optimizations used that are enabled by the tracking of only one symbol instead of two.

Chapter 4

Decoder Architecture

In this chapter we propose an architecture for the major components of a stochastic LDPC decoder. We limit the designed decoders to codes over GF(2^p) as these are the non-binary codes most commonly used in literature and such a constraint greatly simplifies the implementation. In this chapter GF(2^p) is used when describing items which only apply to codes over GF(2^p); while GF(q) is used when there is no restriction on the field order.

There are two common methods for representing GF(2^p) symbols in a circuit: the logarithm representation which uses the discrete logarithm of field elements and since $\log 0$ does not exist, the value 2^p represents 0; and the polynomial representation where coefficients of the element's polynomial are used. The logarithm representation results in simple multiplication circuitry; while the polynomial representation results in simple addition and subtraction circuitry.

We use the polynomial representation for this architecture since it significantly simplifies the check node implementation without a significant increase in the complexity of the permutation node as shown in Sections 4.3 and 4.4.

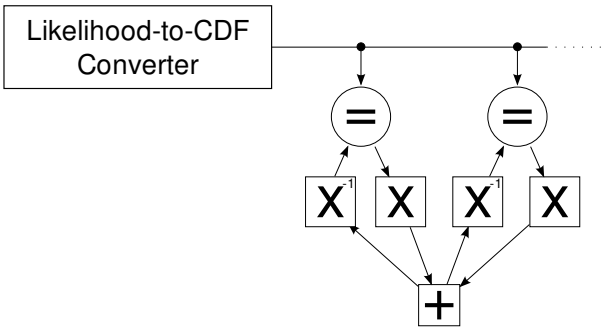


Figure 4–1: Overall decoder architecture

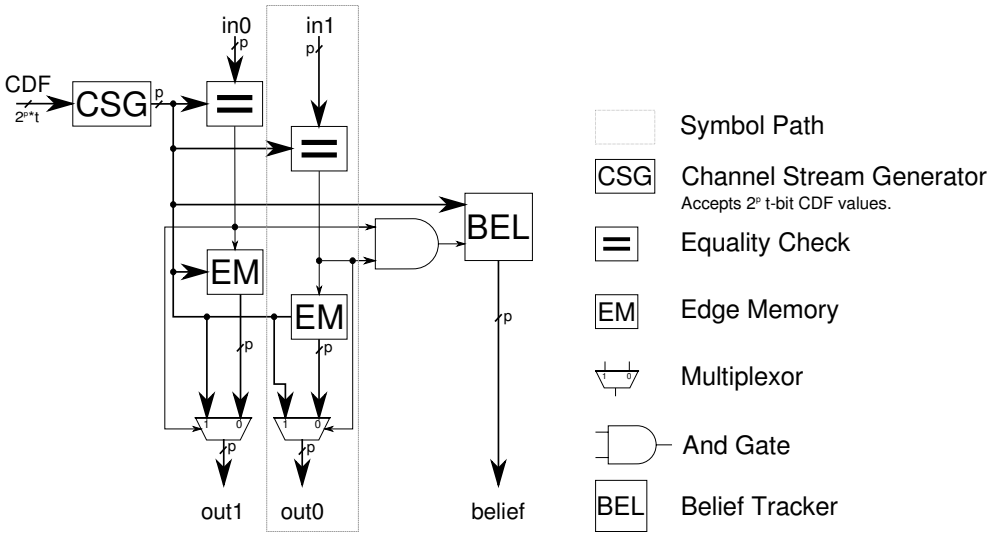
We start this chapter with an architecture overview detailing the location and interconnections of the major decoder blocks in Section 4.1. The details of each block are then presented in Sections 4.2-4.5.

4.1 Architecture Overview

There are four major types of blocks in the stochastic decoder. The likelihood-to-CDF converter receives soft information from the channel and converts it to a cumulative distribution function (CDF) for each variable node. Only one such block is needed in the decoder since data is received serially from the channel. Variable nodes are initialized using the CDF values from the converter. They connect to check nodes via permutation and inverse permutation nodes. Figure 4–1 details the connections between the four block types.

4.2 Variable Node

Figure 4–2a shows the architecture of a degree-2 variable node. For reasons of organizational clarity, this block also includes edge-memories, the channel-stream generator, and the belief tracking circuitry.



(a) Node (b) Legend

Figure 4–2: Degree-2 variable node architecture.

As mentioned in Section 3.3, the variable node components are initialized using the scaled channel likelihood values. This is accomplished by sending a CDF to the channel-stream generator (CSG). Using a cumulative distribution function (CDF), as opposed to a PMF, greatly reduces the complexity of the CSG as shown in Section 4.2.1. The CSG is used to initialize the edge-memories by generating a random stream and writing the values to the edge-memories.

When a symbol arrives on `in0`, it is compared with the CSG output symbol. If both symbols are equal, the value is added to the edge-memory and it is presented as the node output to a permutation node connected to `out1`. On the other hand, if the symbols are not equal, the edge-memory sets the value on `out1`. The symbol coming from `in1` follows a parallel path. When `in0`, `in1`, and the CSG symbol are equal, the node belief is updated as described in Section 4.2.2.

When implementing a higher degree node one needs extra symbol paths identical to the one indicated in Figure 4–2a. The outputs of the additional equality checks are routed to the AND gate at the input of the belief tracker.

4.2.1 Channel-Stream Generator

The CSG's function is to generate a stochastic stream whose PMF is the scaled channel likelihood values. It includes a $\tilde{q}$-bit linear feedback shift-register (LFSR) matching the quantization level of the CDF values. The LFSR generates a uniformly distributed pseudo-random number every cycle which is compared with each CDF value. The first GF(q) symbol whose CDF value is greater than the LFSR output, is added to the channel stream. This can be accomplished using a priority encoder whose inputs are the comparator outputs and whose output is the current channel-stream symbol.

Storing and comparing the LFSR output with the last CDF value is superfluous since that value is always the largest possible at the given quantization level. Thus, it is not stored and the corresponding GF(q) symbol is added to the channel stream when the generated random number is greater than all other CDF values. Figure 4–3 shows an implementation of a GF(4) CSG which only stores $q - 1 = 3$ CDF values.

Had a PMF been used instead of a CDF, an accumulator would have been needed to sum all PMF values until the result exceeded the LFSR output. This would have had to be performed for every decoding cycle, significantly degrading the decoder throughput.

It should be noted that since the LFSR is always enabled and generating random numbers, the LFSRs in all CSG units always provide the same output unless initialized with different seeds. Simulations have shown that a single LFSR for all

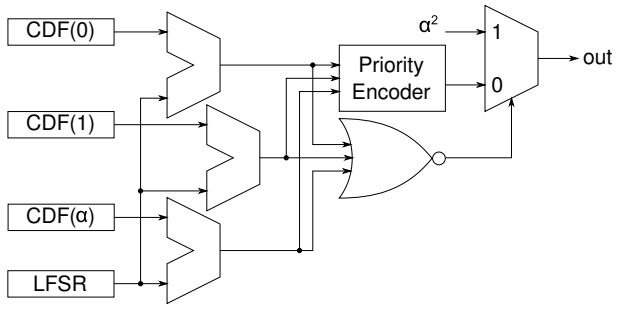


Figure 4–3: GF(4) Channel stream generator

CSGs can be used without a loss in performance. Therefore, multiple CSG blocks can share the same LFSR. The ratio of CSG blocks to LFSRs is a parameter that can be used to favor a lower number of logic gates or simpler chip routing.

4.2.2 Belief Tracker

A variable node's belief is the most likely GF(q) symbol according to all incoming messages. It represents the decoder output for that variable node and is used for calculating the stopping criterion. Deciding which symbol is the belief value is the belief tracker's function.

The tracker contains an up-counter for each GF(q) element. A counter is incremented when all node inputs, including the CSG symbol, are equal to the element it represents. The symbol whose counter has the largest value is the variable node belief.

It is prohibitively expense, in terms of resources used, to find the maximum counter value using a comparator tree. A more efficient approach would be to store the current belief in a register and compare the newly incremented symbol's counter to the belief's. If the new counter is greater than the belief's, the belief register

is updated. Figures 4–4a and 4–4b represent two architectures implementing the aforementioned method for a GF(4) belief tracker.

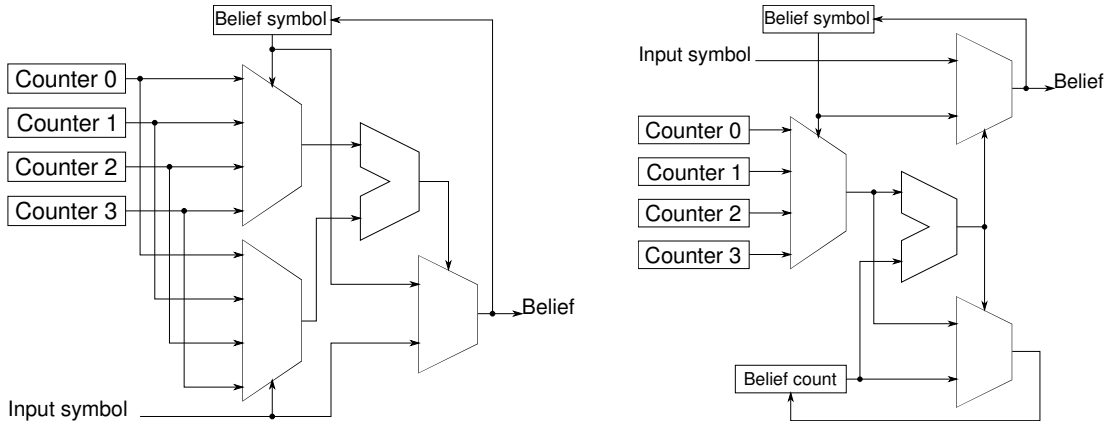
The architecture in Figure 4–4a uses two multiplexers connected to the up-counters. One multiplexer is controlled using the new input symbol; while the other uses the belief register value. The outputs of the multiplexers are compared in magnitude and the symbol corresponding to the larger value is used as the belief value and is stored in the belief register.

The second architecture uses two registers, one for the belief symbol and one for the maximum counter value, and one multiplexer connected to the up-counters and controlled using the new input symbol. The multiplexer output is compared with the maximum count register. The result of that comparison is used to select the larger value, which will be stored in the maximum count register, and the corresponding symbol, which will be stored in the belief register. The belief register is presented as the tracker's output.

Both architectures have similar resource requirements for smaller fields. However as the field size grows, the multiplexers connected to the counters grow significantly in size and complexity. As such, the two register approach is preferable for larger fields.

4.2.3 Edge-Memories

As mentioned in Section 3.2, edge-memories are finite depth buffers, in which a write causes the oldest value to be erased, and a read fetches data from a random location in the memory.



(a) Two multiplexer architecture. (b) Two register architecture

Figure 4–4: GF(4) belief tracker architecture.

Such a system can be implemented using a $\lceil \log_2(q) \rceil$-bit wide FIFO whose read pointer is modified so that it is the output of an LFSR or another random number generator.

The state of the random number generator should be changed only when a write to the edge-memory occurs. Otherwise, all edge-memories use the same read pointer and the correlation between the stochastic message streams is no longer reduced.

4.3 Permutation Node

The GF(q) multiplication by a constant field element can be implemented using either a LUT or a logic multiplier as shown in [25]. Both approaches are equivalent in terms of complexity for GF(2^p), with the LUT being the better for field-programmable logic arrays (FPGA) especially if the field size is small.

We choose to use LUTs as they have a major advantage over the multipliers by constant: they perform the multiplication using a single clock cycles where as the multipliers require multiple cycles since they contain feedback shift-registers.

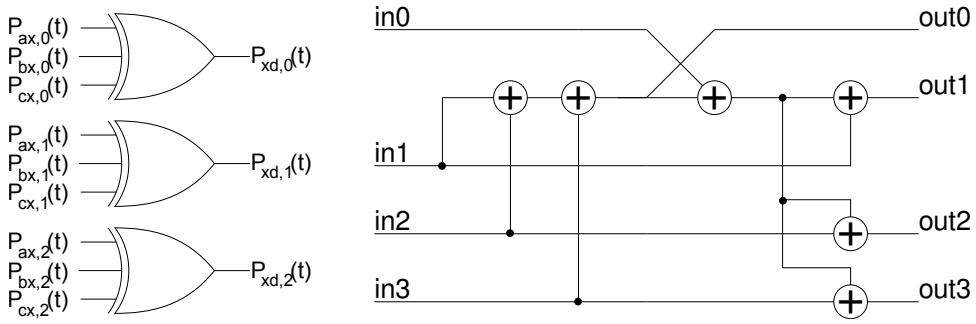
Had the logarithm representation of GF(q) symbols been used, multiplication would have been performed using real addition and subtraction. This is generally less complex than LUTs. However, since the multiplication performed is always by a constant, the LUTs can be kept small.

Let $p = \lceil \log_2(q) \rceil$, then each permutation node requires two $p \times 2^p$ LUTs, one for multiplication by h and the other by h^{-1}. Similarly, two GF(2^p) multipliers would be needed if the combinatorial approach is used.

4.4 Check Node

Since we limit the decoding to GF(2^p), addition and subtraction are the same operation and we will use the term adder to refer to the arithmetic unit which performs this operation. Also, because the polynomial representation of GF(2^p) is used, such an arithmetic unit can be implemented using parallel XORs as shown in Figure 4–5a.

Check nodes are the major reason why the polynomial representation of GF(2^p) elements is used. The logarithm representation would have required the use of LUTs to convert every input into the polynomial representation before performing the addition and another set of LUTs to convert the results back into the logarithm representation; thus, negating the advantage of not using LUTs in the permutation node. Another approach, which might seem appealing at first, is to use LUTs to perform the GF(2^p) addition pairwise. This is not feasible as the size of the LUTs is $2p \times 2^p$, and the latency of going through many of them will be high and will limit the decoder frequency.



(a) 3-input GF(8) adder

(b) $d_c = 4$ check node. $\oplus$ is a GF(2^p) adder.

Figure 4–5: Check node architecture.

Since, regardless of which representation is chosen, LUTs are used, it is best to use the polynomial representation and limit the LUTs to permutation nodes as that approach provides the lowest latency while using the least number of gates.

Computing the outgoing check node messages independently is wasteful of resources as it requires $d_c(d_c - 1)$ GF(2^p) additions per check node. A more efficient approach is to sum all incoming messages together, and when sending a message to permutation node x, subtract the incoming message from x, so that output message is computed using $\overline{V}_{cx}(t) = S - \overline{U}_{xc}(t)$, where $S = \sum_{i=1}^{d_c} \overline{U}_{ic}(t)$. This approach requires only $2(d_c - 1)$[1] GF(2^p) adders as shown in Figure 4–5b.

4.5 Likelihood-to-CDF Converter

The likelihood-to-CDF converter, when operating over GF(2^p), receives soft information regarding the values of bits received from the channel. It computes the likelihood values and applies noise-dependent scaling (NDS) based on the channel model. Then, it proceeds and combines the individual bit likelihood values into

[1] $(d_c - 2) + 1 + (d_c - 1) = 2(d_c - 1)$ [26]

symbol likelihood values. These values are then accumulated to generate a non-normalized CDF, which is then divided by its last, and largest, value resulting in a proper CDF that is sent to a variable node. Since this block requires knowledge of the channel model to compute and combine the likelihood values, we present an architecture for working with the AWGN channel with BPSK modulation.

Computing the NDS bit likelihoods directly involves exponentiation and division which are expensive operations to implement in hardware. Since it was determined via simulation that 6 bits of quantization for these values resulted in no performance loss when compared to floating point simulations, 6×64 LUTs can be used to perform this calculation. For the AWGN channel with BPSK modulation , two LUTs are needed: one to compute the likelihood of the received bit being 0, and the other for the likelihood of it being 1. In addition, using LUTs prevents overflow and saturation. The LUT values are computed offline. For example, when using BPSK modulation over an AWGN channel, the LUTs are populated using: $ce^{-(x-b)^2 \frac{\alpha}{Y}}$ where $c = \frac{2^6-1}{2^6}$ preventing overflow, x is the received bit, $b = 1$ or -1 depending on which LUT is being populated, and $\frac{\alpha}{Y}$ is the NDS factor.

Combining the bit likelihood values to compute the symbol likelihoods is an operation akin to convolution: the likelihood of each bit being 0 or 1 is multiplied by the likelihoods of all other bits being 0 or 1 according to the following equation: $L[\beta] = \prod_{k=1}^{p} l(i_k = \beta_k)$, where $l(i_k = \beta_k)$ is the likelihood of bit k in the received symbol i being equal to bit k in the polynomial representation of GF(2^p) symbol β.

The symbol likelihood values are a non-normalized PMF. Since it was shown in Section 4.2.1 that it is beneficial to use CDFs, the values in the non-normalized PMF are accumulated such that $c\text{CDF}[i] = \sum_{k=0}^{k=i} c\text{PMF}[k]$ where cCDF and cPMF are the

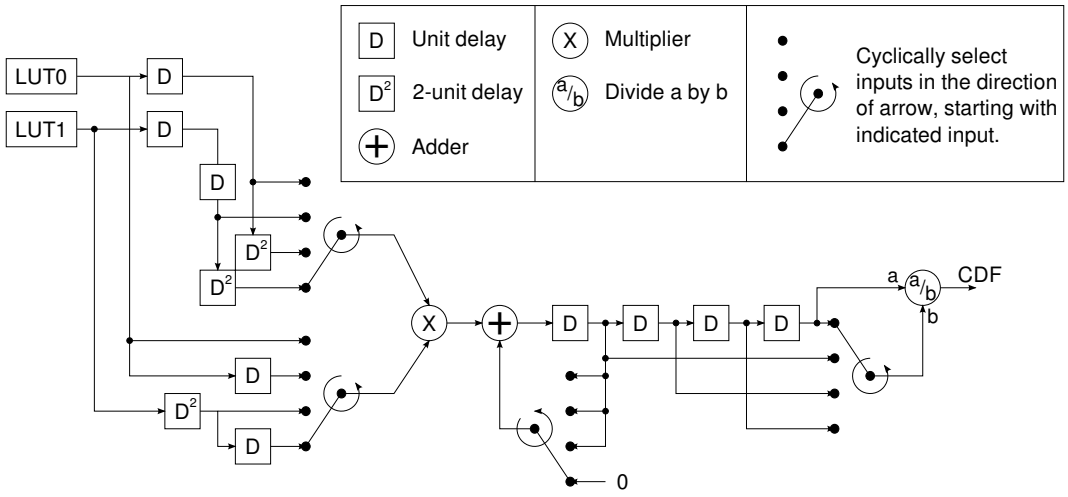


Figure 4–6: Pipelined GF(4) likelihood-to-CDF converter

non-normalized distribution functions. The received symbol CDF is computed using $\text{CDF}[i] = c\text{CDF}[i]/c\text{CDF}[2^p - 1]$, where $i = 0$ to $2^p - 1$.

Figure 4–6 demonstrates a pipelined likelihood-to-CDF converter designed for GF(4) and transmission over the AWGN channel with BPSK modulation.

4.6 Complexity Analysis

In this Section, we compare the number of operations performed per decoding cycle by the proposed stochastic decoder with that of the FFT-SPA and Log-FFT-SPA decoders of [3]. EMS and Reduced complexity EMS are not included in this comparison as they focus on a subset of the GF(2^p) symbols and we limit the analysis to algorithms that include the entire field.

We start with the variable node which performs the most varied types of operations. Every decoding cycles, a variable node performs d_v equality checks between the CSG output and its inputs. The results of theses checks are combined using AND

gates. The CSG performs $2^p - 1$ memory accesses and comparisons when generating a symbol. To update the node belief, the belief tracker reads the counter value corresponding to the input symbol and increments it, which involves a real addition and a memory write. The registers containing the maximum count and the belief symbol are read and updated after the maximum count is compared with the new symbol count. Therefore, the belief tracker performs 1 real addition, 1 comparison, and 6 memory operations. Each of the d_v edge-memories performs either a write or a read. As such the total number of operations performed by each variable node per decoding cycle is: $d_v + 2^p$ comparisons, $2^p + d_v + 5$ memory accesses, and 1 real addition.

Each permutation node performs two GF(2^p) multiplications, and each check node performs $2d_c - 2$ GF(2^p) additions.

Table 4–1 summarizes the comparison between the two SPA decoders and two versions of the stochastic decoder: one which performs GF(2^p) multiplication using GF(2^p) multipliers by constants, and one which uses look-up tables (LUT). The decoders compared are for a code with n_v variable nodes and n_c check nodes. In addition to arithmetic operations, the table shows equality checks and magnitude comparisons in the logical comparison column, and LUT and register accesses in the memory access column.

It is evident that stochastic decoding performs far fewer and simpler operations per iteration than SPA based decoders. The one drawback it has is the large number of iterations which is addressed in Chapter 5.

Algorithm	Multiplication		Addition		Logical Comparison	Memory Access
	Real	GF(2^p)	Real	GF(2^p)		
FFT-SPA [3]	$2^p(d_v^2+4d_v)n_v$	0	$(2pd_v+1)2^pn_v$	0	N/A	0
Log-FFT-SPA [3]	0	0	$(2p+4)2^pd_vn_v$	0	N/A	$2p2^pd_vn_v$
Stochastic	0	$2d_vn_v$	n_v	$(2d_c-2)n_c$	$(d_v+2^p)n_v$	$(2^p+d_v+5)n_v$
Stochastic-LUT	0	0	n_v	$(2d_c-2)n_c$	$(d_v+2^p)n_v$	$(2^p+3d_v+5)n_v$

Table 4–1: Total number of operations per iteration.

4.7 Synthesis Results

The nodes were synthesized for the (256,128) GF(16) LDPC codes in simulations using a 90nm library with a target frequency of 100MHz.

The variable nodes were by far the largest with each variable node occupying 30338 μm^2. The LDPC code used an equal number of four different values for permutation node multipliers: $\{1, \alpha^3, \alpha^7, \alpha^{11}\}$. This resulted in permutation nodes with different sizes since the synthesizer implemented the LUTs as combinatorial logic and was able to optimize some better than the others. The permutation node with multiplier 1 does not occupy any area since it is just a connection between the variable and check nodes. The node with α^3 occupied 71 μm^2, that with α^7 also occupied 71 μm^2, and the node with α^{11} occupied 78 μm^2. These estimates include the forward and inverse permutation node LUTs. Each check node required 158 μm^2.

These area estimates are for synthesized logic only, routed area was not calculated. Control logic area was not computed as it would be minuscule compared to the node area. Also, The likelihood-to-CDF was not include either for the same reason.

	Var. Nodes	Perm. Nodes	Check Nodes	Total
Area (μm^2)	7,766,528	28,160	20,224	7,814,912
Gate Count (1× NAND)	1,769,141	6,414	4607	1,780,162

Table 4–2: Synthesis results

Table 4–2 shows the area and estimated gate count for all nodes in the decoder in addition to an estimated total.

The synthesis results show that variable nodes occupy most (99%) of the decoder area. As can be seen from Table 4–3, the majority of the variable node area is composed of the EMs and CSG. Therefore, any scheme which can simplify these two components or reduce their count can significantly decrease the decoder area.

	EMs	CSG	Belief Tracker	Other	Total
Area (μm^2)	15023	9693	5419	203	30338
% of node area	49.5	32.0	17.9	0.6	100

Table 4–3: Variable node area use

Chapter 5

Reducing the Number of Decoding Cycles

In section 3.4, we saw that the average and maximum number of decoding cycles for the stochastic decoder were large compared with those of the SPA decoder. This limits the decoder applications since it decreases throughput and increases maximum latency.

In this chapter, we study two methods for reducing the number of decoding cycles that were first used for binary stochastic LDPC decoding in [27] and [28]. In sections 5.1 and 5.2 we generalize these two techniques to GF(q) decoders and investigate their performance and some simplifications which reduce the resources needed.

5.1 Tracking Forecast Memories

Tracking forecast memories (TFM) were first introduced in [27] as a replacement for edge-memories in order to reduce chip area. While they do not necessarily reduce area requirements for LDPC codes over GF(q), they significantly increase the rate at which the decoder converges.

TFMs replace edge-memories in the decoder without further changes to its architecture. They modify the variable node output using the successive relaxation method [29]. A TFM operating over GF(q) contains q registers storing a PMF which corresponds to the variable node output. When a GF(q) symbol x is written to a TFM at time t, each PMF register is updated according to the following rule:

$$\mathrm{PMF}_t[i] = \begin{cases} (1-\beta)\mathrm{PMF}_{t-1}[i] + \beta & \text{if } i = x \\ (1-\beta)\mathrm{PMF}_{t-1}[i] & \text{otherwise} \end{cases} \tag{5.1}$$

In equation (5.1), $i \in GF(q)$ and $\beta \leq 1$ is the relaxation factor. β is computed offline via simulation and values of the form 2^{-k}, $k \in \mathbb{N}$ are desirable since they result in simpler hardware implementation as will be explained in section 5.1.1. The PMF values are used to generate the TFM output symbol when a read is requested.

Figure 5–1 demonstrates that a TFM-based decoder with $\mathrm{DC}_{\max} = 10^4$ outperforms a regular EM-based one operating with $\mathrm{DC}_{\max} = 10^5$, and the TFM-based decoder with $\mathrm{DC}_{\max} = 10^6$ matches the SPA curve. Table 5–1 shows that this performance is achieved with a significantly lower average cycle count. Further reduction in $\mathrm{DC}_{\max}$ was not possible as the decoder performance was extremely degraded when $\mathrm{DC}_{\max} = 10^3$. This is to be expected since the average number of iterations is larger than or very close to 10^3 for the SNR range simulated. All the TFM-based decoders used in the examples use $\beta = 1/16$.

5.1.1 TFM Architecture

From equation (5.1) it is evident that the PMF values stored in a GF(q) TFM are updated independently of each other. This forms the basis of the block's architecture: a TFM is divided into sub-blocks, called TFM[i], where $i \in$ GF(q). Each TFM[i]

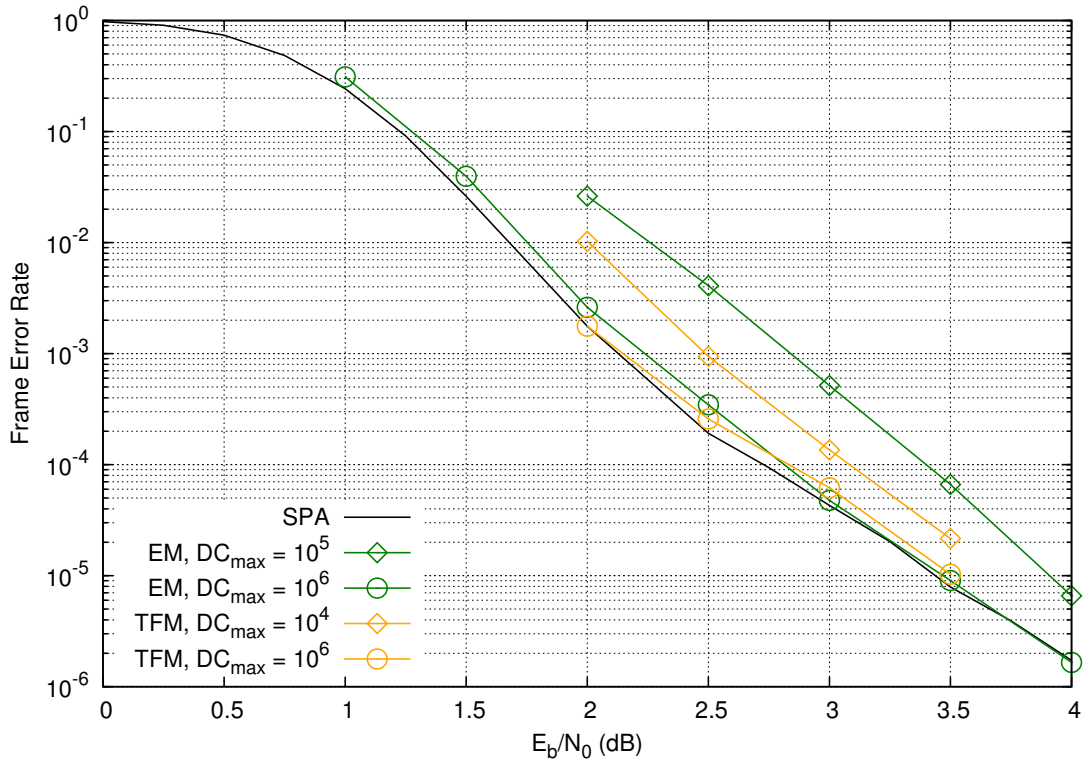


Figure 5–1: TFM-based decoder performance

tracks the PMF value associated with i and provides that value as output. The outputs are used as a PMF to generate a discrete random variable that is the TFM output symbol. Figure 5–2 shows the architecture of a GF(q) TFM.

In the block diagram of a TFM[i], we notice that the multiplication of the PMF[i] value by $(1-\beta)$ is implemented as PMF[i] $-\,\beta * $PMF[$i$]. This eliminates the need for a real multiplier circuit if $\beta = 2^{-k}$ since the multiplication can be simply accomplished by shifting the bits of PMF[i] by k positions to the right

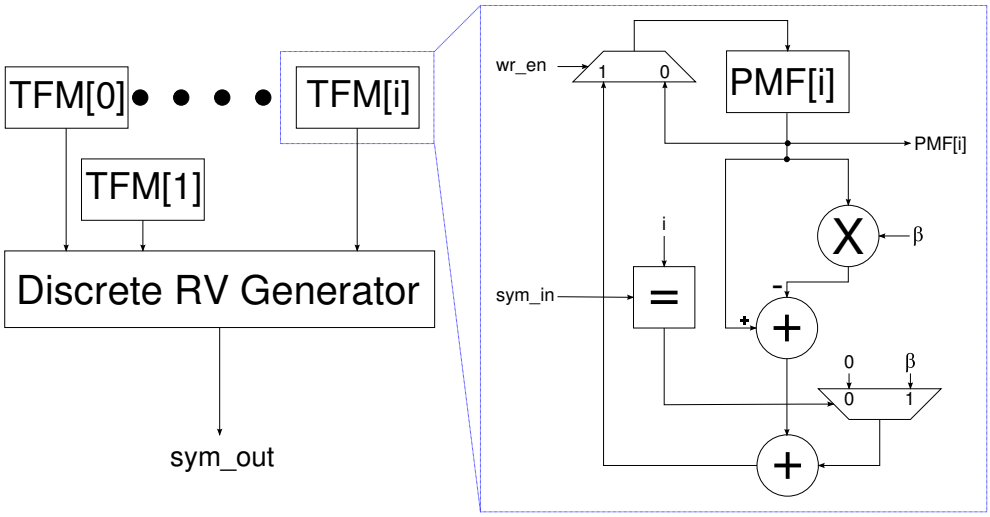


Figure 5–2: TFM architecture

5.1.2 Reducing TFM Count

It was noted in simulations that a number of TFMs in the decoder can be replaced by registers that only store the last written value, i.e. directly implementing equation (3.1), without a major degradation in performance. Figure 5–3 shows the resulting decoder performance when every 2nd (50% TFM), 3rd (66% TFM), or 4th (75% TFM) TFM is replaced with a register compared with the original decoder (100% TFM). In these simulations $\beta = 1/16$ and $DC_{max} = 10^4$.

Since TFMs contain many registers and, thus, occupy a large portion of the decoder, reducing their count would reduce the decoder area significantly.

5.2 Relaxed Half-Stochastic Decoding

Relaxed Half-Stochastic (RHS) decoding, first introduced in [28], stemmed from the observation that the variable node constraint (3.1) is not frequently satisfied in binary stochastic decoding, especially as the node degree increases. As such, it

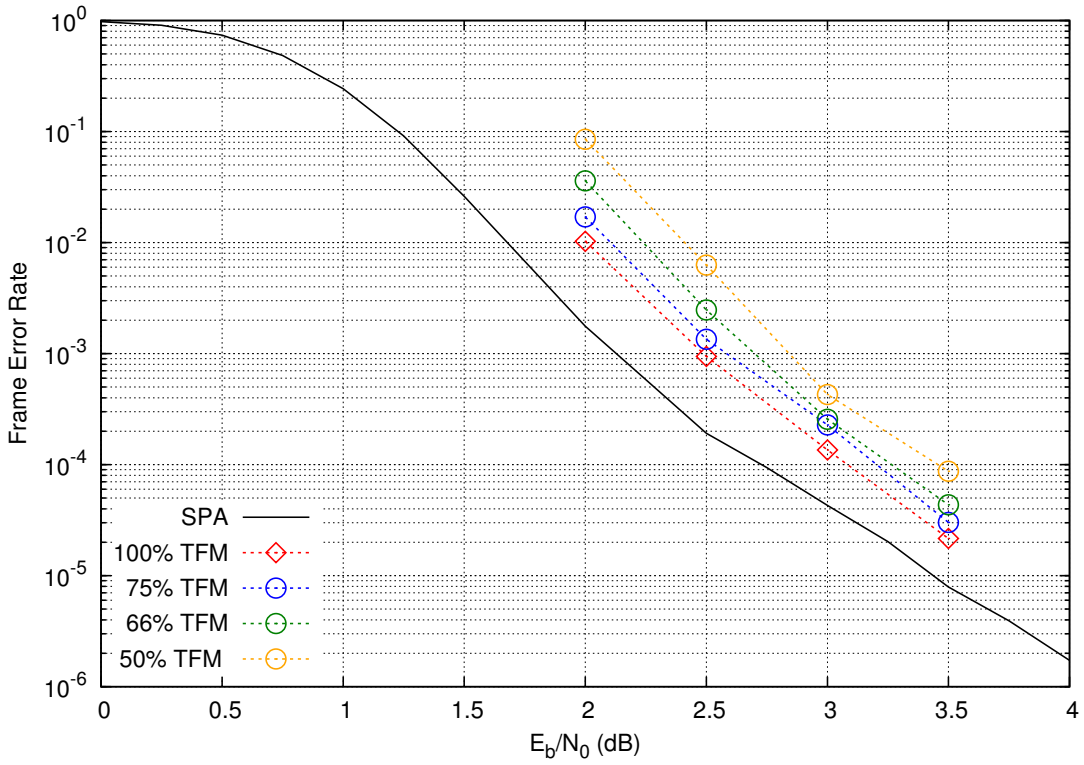


Figure 5–3: Reduced TFM decoder performance, $\mathrm{DC}_{\max} = 10^4$

increases the number of decoding cycles required for decoder convergence. Going to higher field orders exacerbates this issue. To solve this problem Leduc-Primeau *et al.* [28], replaced the stochastic variable node with an SPA one. The SPA variable node has two major advantages: first, it modifies its output message statistics every cycle, not just when the update constraint is satisfied; second, it eliminates the correlation between stochastic streams. The node has a TFM on each input to convert the stochastic streams into probability values, and it uses the SPA output message as a PMF to generate a stochastic stream symbol.

The same approach can be used for LDPC codes over GF(q). A GF(q) SPA variable node is connected to the output of the inverse permutation nodes using

SNR (dB)	2.0	2.5	3.0	3.5
EM ($DC_{\max} = 10^6$), $L = 50$	22599	8888	4243	2329
TFM ($DC_{\max} = 10^6$), $\beta = 1/16$	2923	1141	731	522
TFM ($DC_{\max} = 10^4$), $\beta = 1/16$	1893	1100	730	522
RHS ($DC_{\max} = 10^6$), $\beta = 1/64$	641	240	167	129
RHS ($DC_{\max} = 10^4$), $\beta = 1/64$	373	228	167	129
RHS ($DC_{\max} = 10^3$), $\beta = 1/64$	331	225	166	129

Table 5–1: Average decoding cycles for modified decoders

modified GF(q) TFMs. The input of these TFMs is the inverse permutation node output symbol; however, unlike normal TFMs, their output is the stored PMF values instead of a GF(q) symbol. A variable node output message calculated using (2.9) is used as a PMF to generate the symbol sent to a permutation node. One can use logarithms to convert the multiplications in equation (2.9) to additions, and since a variable node output message is used as a PMF, it must be properly normalized.

From Table 5–1 and Figure 5–4, a conclusion similar to that of the TFM-based decoder case arises; namely that the RHS decoder requires significantly fewer cycles to converge. It also outperforms the TFM-base decoder in both metrics. An interesting result is that the RHS decoder with $\mathrm{DC}_{\max} = 10^3$ outperforms the EM-based one with $\mathrm{DC}_{\max} = 10^5$.

NDS degrades the RHS decoder's performance; therefore, channel likelihood values should not be scaled, just like in an SPA decoder. Also, an RHS variable node's belief is computed in the same manner as in the SPA variable node.

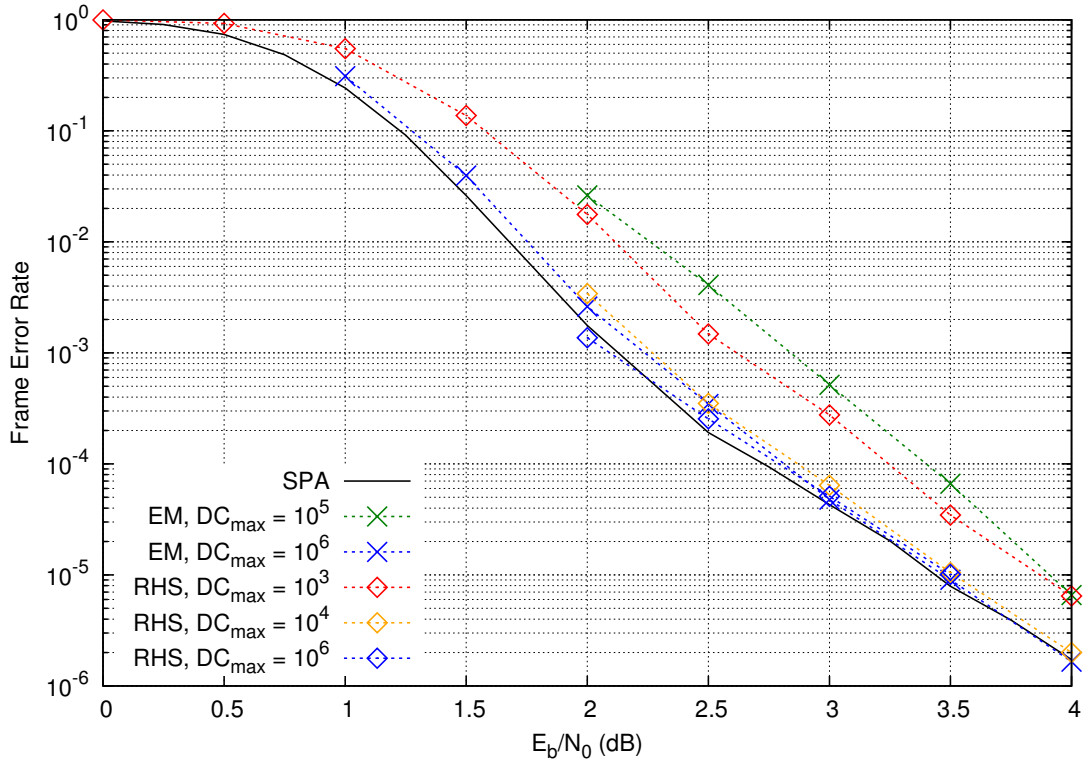


Figure 5–4: RHS decoder performance, $\beta = 1/64$

Chapter 6

Conclusion

In this thesis we presented an algorithm for decoding LDPC codes over GF(q) which has much lower complexity than its counterparts in literature. We demonstrated that its performance matches that of the SPA.

A decoder architecture based on the stochastic decoding algorithm that only uses simple circuitry to compute node equations was provided, and an estimate of the area required for its implementation was given. Analysis of synthesis results showed that edge-memories were occupying a large area.

To increase the decoder throughput two enhancements were studied: TFMs and RHS decoding. We showed that using either of these methods decreased the average number of decoding cycles significantly and, thus, increased throughput and enabled us to lower the decoder's maximum latency. We also investigated an area reduction scheme that removed some TFMs from the system and showed that the resulting performance degradation was within tolerable limits.

Further work on this topic would include reducing the complexity of the TFM implementation, investigating whether the belief tracker can be simplified by using

the EMs or TFMs already present in the node to replace the counters, and improving the decoder performance for higher field orders.

REFERENCES

[1] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 47, no. 2, pp. 619–637, 2001.

[2] M. Davey and D. MacKay, "Low-density parity check codes over GF(q)," *IEEE Commun. Lett.*, vol. 2, no. 6, pp. 165–167, 1998.

[3] H. Song and J. Cruz, "Reduced-complexity decoding of Q-ary LDPC codes for magnetic recording," *IEEE Trans. Magn.*, vol. 39, no. 2, pp. 1081–1087, 2003.

[4] I. Djordjevic and B. Vasic, "Nonbinary LDPC codes for optical communication systems," *IEEE Photonics Technology Letters*, vol. 17, no. 10, pp. 2224–2226, 2005.

[5] G. Sarkis, S. Mannor, and W. J. Gross, "Stochastic decoding of LDPC codes over GF(q)," in *Proc. IEEE International Conference on Communications ICC '09*, Jun. 14–18, 2009, pp. 1–5.

[6] R. G. Gallager, *Low-Density Parity-Check Codes*. Cambridge, MA: MIT Press, 1963.

[7] D. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inf. Theory*, vol. 45, no. 2, pp. 399–431, 1999.

[8] S. Le Goff, A. Glavieux, and C. Berrou, "Turbo-codes and high spectral efficiency modulation," in *Serving Humanity Through Communications Communications ICC 94, SUPERCOMM/ICC '94, Conference Record IEEE International Conference on*, May 1–5, 1994, pp. 645–649.

[9] C. Schlegel and L. Perez, *Terllis and Turbo Coding*. Wiley-IEEE Press, 2003.

[10] D. Declercq and M. Fossorier, "Decoding algorithms for nonbinary LDPC codes over GF(q)," *IEEE Trans. Commun.*, vol. 55, no. 4, pp. 633–643, 2007.

[11] J. Chen, L. Wang, and Y. Li, "Performance comparison between non-binary LDPC codes and Reed-Solomon codes over noise bursts channels," in *Proc.*

International Conference on Communications, Circuits and Systems, L. Wang, Ed., vol. 1, 2005, pp. 1–4 Vol. 1.

[12] L. Barnault and D. Declercq, "Fast decoding algorithm for LDPC over GF(2^q)," in *Proc. IEEE Information Theory Workshop*, Mar. 31–Apr. 4, 2003, pp. 70–73.

[13] D. MacKay and M. Davey, "Evaluation of Gallager codes for short block length and high rate applications," in *Proc. IMA Workshop Codes, Syst., Graphical Models*, 1999.

[14] X. Li and M. R. Soleymani, "A proof of the Hadamard transform decoding of the belief propagation algorithm for LDPCC over gf(q)," in *Proc. VTC2004-Fall Vehicular Technology Conference 2004 IEEE 60th*, vol. 4, Sep. 26–29, 2004, pp. 2518–2519.

[15] H. Wymeersch, H. Steendam, and M. Moeneclaey, "Log-domain decoding of LDPC codes over GF(q)," in *Proc. IEEE International Conference on Communications*, H. Steendam, Ed., vol. 2, 2004, pp. 772–776 Vol.2.

[16] X. Huang, S. Ding, Z. Yang, and Y. Wu, "Fast min-sum algorithms for decoding of LDPC over GF(q)," in *Proc. ITW '06 Chengdu Information Theory Workshop IEEE*, S. Ding, Ed., 2006, pp. 96–99.

[17] A. Voicila, D. Declercq, F. Verdier, M. Fossorier, and P. Urard, "Low-complexity, low-memory EMS algorithm for non-binary LDPC codes," in *Proc. IEEE International Conference on Communications ICC '07*, D. Declercq, Ed., 2007, pp. 671–676.

[18] B. Gaines, *Advances in Information Systems Science*. Plenum, New York, 1969, ch. 2, pp. 37–172.

[19] S. Sharifi Tehrani, W. Gross, and S. Mannor, "Stochastic decoding of LDPC codes," *IEEE Commun. Lett.*, vol. 10, no. 10, pp. 716–718, 2006.

[20] V. Gaudet and A. Rapley, "Iterative decoding using stochastic computation," *Electronics Letters*, vol. 39, no. 3, pp. 299–301, Feb. 2003.

[21] C. Winstead, V. Gaudet, A. Rapley, and C. Schlegel, "Stochastic iterative decoders," in *Proc. International Symposium on Information Theory ISIT*, 2005, pp. 1116–1120.

[22] S. Sharifi Tehrani, S. Mannor, and W. J. Gross, "Fully parallel stochastic LDPC decoders," *IEEE Trans. Signal Process.*, vol. 56, no. 11, pp. 5692–5703, Nov. 2008.

[23] C. Poulliat, M. Fossorier, and D. Declercq, "Design of regular (2, d_c)-LDPC codes over GF(q) using their binary images," *IEEE Trans. Commun.*, vol. 56, no. 10, pp. 1626–1635, October 2008.

[24] A. Voicila, F. Verdier, D. Declercq, M. Fossorier, and P. Urard, "Architecture of a low-complexity non-binary LDPC decoder for high order fields," in *Proc. International Symposium on Communications and Information Technologies ISCIT '07*, F. Verdier, Ed., 2007, pp. 1201–1206.

[25] S. Lin and D. J. Costello, *Error Control Coding*. Prentice Hall, 2004.

[26] J. L. Fan, "Constrained coding and soft iterative decoding for storage," Ph.D. dissertation, Stanford University, Stanford, CA, December 1999.

[27] S. Sharifi Tehrani, A. Naderi, G.-A. Kamendje, S. Mannor, and W. J. Gross, "Tracking forecast memories in stochastic decoders," in *Proc. IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP 2009*, Apr. 19–24, 2009, pp. 561–564.

[28] F. Leduc-Primeau, S. Hemati, W. Gross, and S. Mannor, "A relaxed half-stochastic iterative decoder for LDPC codes," to appear in the Proceedings of IEEE Globecom 2009.

[29] S. Hemati and A. H. Banihashemi, "Dynamics and performance analysis of analog iterative decoding for low-density parity-check (LDPC) codes," *IEEE Trans. Commun.*, vol. 54, no. 1, pp. 61–70, Jan. 2006.