This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

Adaptive Multiset Stochastic Decoding of Non-binary LDPC Codes

Alexandru Ciobanu, Saied Hemati, *Senior Member, IEEE*, Warren J. Gross, *Senior Member, IEEE*

***Abstract*—We propose a non-binary stochastic decoding algorithm for low-density parity-check (LDPC) codes over GF(q) with degree two variable nodes, called Adaptive Multiset Stochastic Algorithm (AMSA). The algorithm uses multisets, an extension of sets that allows multiple occurrences of an element, to represent probability mass functions that simplifies the structure of the variable nodes. The run-time complexity of one decoding cycle using AMSA is $O(q)$ for conventional memory architectures, and $O(1)$ if a custom memory architecture is used. Two fully-parallel AMSA decoders are implemented on FPGA for two (192,96) (2,4)-regular codes over GF(64) and GF(256), both achieving a maximum clock frequency of 108 MHz. The GF(64) decoder has a coded throughput of 65 Mbit/s at $E_b/N_0 = 2.4$ dB when using conventional memory, while a decoder using the custom memory version can achieve 698 Mbit/s at the same E_b/N_0. At a frame error rate (FER) of 2×10^{-6} the GF(64) version of the algorithm is only 0.04 dB away from the floating-point SPA performance, and for the GF(256) code the difference is 0.2 dB. To the best of our knowledge, this is the first fully-parallel non-binary LDPC decoder over GF(256) reported in the literature.**

***Index Terms*—Iterative decoding, low-density parity-check code, stochastic decoding, parallel architectures, non-binary codes.**

I. Introduction

INTRODUCED in 1962 by Gallager [1] and rediscovered by MacKay three decades later [2], low-density parity-check (LDPC) codes are linear error-correcting block codes built using sparse bipartite graphs.

LDPC codes are widely recognized for the channel capacity-approaching performance [2], [3], [4] and the high degree of parallelism in decoding operations. These properties led to the inclusion of LDPC codes in multiple communications standards like DVB-S2 (satellite broadband) [5], ITU-T G.hn (networking over power lines, phone lines, and coaxial cable) [6], IEEE 802.3an (10GBase-T Ethernet) [7], IEEE 802.11n-2009 (Wi-Fi) [8], IEEE 802.16e (WiMAX) [9], and others. Additionally, LDPC codes have found their use in storage systems [10], [11].

Non-binary LDPC codes have been shown to have better resilience against burst errors [11] and mixed types of noise and interference [12], are better suited for higher-order modulation, and provide a considerable performance boost to medium and short length codes. Because of these properties non-binary LDPC codes were studied within the DAVINCI project which aims at further reducing the gap between state-of-the-art performance of practical codes and Shannon capacity [13], [14].

The performance improvements associated with the generalization of LDPC codes to non-binary GF(q) fields comes at a cost. The complexity of the Sum-Product Algorithm (SPA) under GF(q) becomes $O(q^2)$, which limits the feasibility of non-binary decoders to lower-order GF(q) fields. There have been multiple attempts at tackling the complexity problem with algorithms like FFT-SPA [15], Log-SPA [16], and Extended Min-Sum [17] (EMS) but the complexity remains high and a fully-parallel implementation of these decoders has not been shown to be practical to date. The highest previously reported throughput for a hardware implementation of the GF(64) code used in this work is of 3.8 Mbit/s [18].

Stochastic decoding for LDPC codes was introduced in [19] as a way of reducing hardware complexity [20] while matching and even improving on the performance of reference algorithms like SPA in both binary [19], [21], [22] and non-binary [23] cases. In stochastic decoding, instead of the probabilities of symbols the symbols themselves are sent as messages with the probabilities being encoded in the statistics of the stream. Despite the implementation advantages of non-binary stochastic decoders designing a fully-parallel decoder remains challenging, especially for high-order fields GF($q \geq 64$).

In this paper, we present a new stochastic decoding algorithm for non-binary LDPC codes that considerably reduces the complexity of the computations performed in the variable nodes (VNs) while inheriting the benefits of very simple check nodes (CNs) and interleaver circuit [20], [24]. The algorithm is called Adaptive Multiset Stochastic Algorithm (AMSA).

Additionally, this work successfully extends the redecoding technique [22] to non-binary codes. Redecoding is a technique, introduced originally for relaxed half-stochastic (RHS) decoding of binary LDPC codes, that improves bit-error-rate (BER) performance and lowers error floors by making multiple decoding attempts on codewords that fail to decode initially.

Furthermore, we propose a fully-parallel architecture for AMSA, which is used for implementing LDPC decoders for two practical codes from the DAVINCI project [13], [14]. The GF(64) decoder achieves a coded throughput of 65 Mbit/s using conventional memory and 698 Mbit/s using custom memory at an SNR of 2.4 dB, which are, to the best of our knowledge, the highest reported throughput for this code. We show that AMSA decoder architecture scales gracefully with the order of the field q. The length of the memory blocks

Copyright (c) 2012 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained from the IEEE by sending a request to pubs-permissions@ieee.org.

A. Ciobanu was at the time of writing with the Department of the Electrical and Computer Engineering, McGill University, Montreal, QC H3A 2A7, Canada, currently with Rogue Research Inc., Montreal, QC, Canada (e-mail: alexandru.ciobanu@mail.mcgill.ca). S. Hemati is with the Department of Electrical Engineering, Linkoping University, Linkoping, Sweden (e-mail: saied.hemati@liu.se). W. J. Gross is with the Department of the Electrical and Computer Engineering, McGill University, Montreal, QC H3A 2A7, Canada (e-mail: warren.gross@mcgill.ca).

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

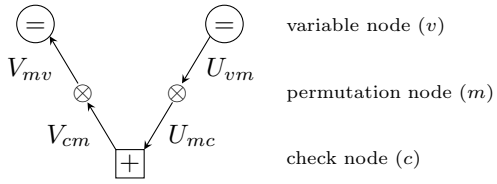

Fig. 1. Types of nodes in non-binary SPA decoder.

scales with $O(q)$ while the width of the memory blocks, the number of wires in the decoder, the length of the registers, and the size of control logic scale with $O(\log q)$. Moreover, we are not aware of any other reported fully-parallel non-binary LDPC decoder implementation for GF(256).

The term conventional memory is used in this paper to refer to memory architectures where the task of writing k identical words takes a number of steps that is proportional to k.

A background on LDPC decoding and relevant topics is given in Section II. The Adaptive Multiset Stochastic Algorithm and its hardware implementation are presented in Sections III and IV. Section V is concerned with the analysis of the simulation results. Finally, the conclusion is found in Section VI.

II. LDPC Decoding over GF(q)

A. SPA Decoding Over GF(q)

SPA was originally extended to non-binary LDPC codes over GF(2^p) in [29], and in [15] it was shown that the check node computations can be simplified by introducing permutation nodes on the edges of the Tanner graph as shown in Figure 1. These additional nodes perform a permutation on the pmf messages exchanged between the variable and check nodes. Note that the subscripts in the message names specify the source and destination nodes of the messages, e.g. U_{vm} is a message from variable node v to permutation node m. Using this notation, the SPA algorithm for non-binary LDPC codes can be expressed in the following way:

1) *Initialization*
Each variable node v is initialized with the channel likelihoods $L_v[i_1..i_p] = \prod_{l=1}^{p} \Pr(b_l = i_l \mid y_l)$ where $L_v[i_1..i_p]$ is the channel likelihood of the GF(2^p) symbol specified by the bits $(i_1..i_p)$, b_l is the lth bit of the symbol, and y_l is the corresponding noisy bit received.
2) *Product step*
For each variable node v the output messages U_{vt} are computed for all the symbols in GF(2^p), i.e. for all $(i_1..i_p) \in \{0,1\}^p$:

$$U_{vt}[i_1..i_p] = L_v[i_1..i_p] \prod_{m=1, m\neq t}^{d_v} V_{mv}[i_1..i_p] \quad (1)$$

where $t = 1, \ldots, d_v$, and $L_v[i_1..i_p]$ are the channel likelihoods computed in the initialization step, V_{mv} are the variable node incoming messages, and all the products are tensor dot products. Note that after this step U_{vt} requires normalization so that $\sum_{i_1..i_p} U_{vt}[i_1..i_p] = 1$.
3) *Permutation step*
A Tanner graph edge between a variable node v and a check node c corresponds to a non-zero element h_{vc} in the H matrix of the non-binary code. The output of this step is a pmf message U_{mc} which is a permutation of the input message U_{vm}:

$$U_{mc}[i_1..i_p] = U_{vm}[j_1..j_p], \quad (2)$$

where $(i_1..i_p)$ and $(j_1..j_p)$ bits correspond to GF(2^p) symbols α_i and α_j respectively, such that $\alpha_i = \alpha_j h_{vc}$. Passing a message on the same edge in the opposite direction, from a check node c to a variable node v, involves the inverse permutation given by $h_{cv} = h_{vc}^{-1}$.
4) *Check step*
The sum-product update computes the output messages V_{ct} corresponding to a check node c:

$$V_{ct}[i_1..i_p] = \sum_{(j_1..j_p)\in C_t[i_1..i_p]} \prod_{m=1, m\neq t}^{d_c} U_{mc}[j_1..j_p] \quad (3)$$

where $C_t[i_1..i_p]$ is the set of symbols a_j corresponding to bits $(j_1..j_p)$ such that $\sum \alpha_j = \alpha_i$ over GF(2^p), where α_i is the symbol corresponding to bits $(i_1..i_p)$, and $t = 1, \ldots, d_c$.

The complexity of one decoding iteration for SPA is $O(Ntq^2)$, where N is the codeword length and t is the average column weight in matrix H, and is dominated by the complexity of the sum-product computation.

Log-SPA, the logarithm domain variation of the SPA algorithm [30], converts the multiplication operations into additions, but does not reduce the computational complexity. The FFT-SPA algorithm [11], [31] performs the check node computations in frequency domain and reduces the complexity to $O(Mqu(\log q + u))$, where M is the number of mutually independent rows in H, and u is the mean row weight. The Extended Min-Sum Algorithm (EMS) [15] decreases the size of the non-binary messages from q to $n_m \leq q$ and reduces the complexity to $O(n_m \log n_m)$ at the cost of some decoding performance loss.

B. Stochastic Decoding Over GF(q)

Stochastic decoding is inspired by the technique of stochastic computing [32] where quantities are represented as Bernoulli sequences of bits and the information is conveyed through the statistics of the stream. This stream representation allows for complex computations to be implemented with simple hardware, and reduces the number of interconnecting wires required. The result of these reductions in complexity are circuits that can sustain higher clock rates [32].

The stochastic concept can be extended to the non-binary case where a stochastic stream of symbols can be used to represent the distribution of probabilities of the symbols. Let $\alpha, \beta, \gamma, \delta$ be the four possible symbols in GF(4). The non-binary stream $s_{nb} = \gamma\beta\alpha\alpha\gamma\beta\delta\alpha\gamma\alpha\alpha\gamma\ldots$ is equivalent to the following distribution of probabilities: $p_\alpha = \frac{5}{12} \approx 0.416$, $p_\beta = \frac{2}{12} \approx 0.166$, $p_\gamma = \frac{4}{12} \approx 0.333$, $p_\delta = \frac{1}{12} \approx 0.083$. The distributions created this way are always normalized because $\sum_{x\in GF(q)} n_x = n_t$ where n_x is the

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

TABLE I
GF(q) LDPC DECODERS IN LITERATURE

Implementation	[25]	[26]	[27]	[18]	[28]
Galois Field	GF(4)	GF(8)	GF(32)	GF(64)	GF(64)
Code	$N = 486$ $K = 972$	$N = 720$	$N = 620$ $K = 310$	$N = 192$ $K = 96$ (2,4)-regular	$N = 160$ $K = 80$ (2,4)-regular
Type of architecture	partially-parallel	serial	partially-parallel	serial	fully-parallel
Max. decoding iterations	not reported	15	10	8	30
Platform	FPGA	FPGA	ASIC	FPGA	ASIC
Frequency	131.4 MHz	99.73 MHz	200 MHz	75 MHz	700 MHz
Throughput	50 Mbit/s	1.09 Mbit/s	60 Mbit/s	3.8 Mbit/s	1.15 Gbit/s

number of times symbol x has been observed and n_t is the total number of symbols.

It is important to note the difference in the nature of the messages in SPA or MSA, and the ones used in stochastic decoding. In stochastic decoding a message consists of a single symbol from GF(q) [19]. The probability distribution associated with the stochastic stream can be inferred as shown above. In contrast, in SPA and MSA a message is an expression of a distribution of probabilities, a probability mass function (pmf) represented by a vector of length q, while in EMS the length is reduced to $n_m \leq q$ [15].

The amount of information needed to represent an SPA or MSA message is qw bits (or $n_m w$ bits for EMS) where w is the number of bits required to store each probability or likelihood. The number of bits needed to represent a stochastic message is $\log q$, making them considerably more compact than their SPA, MSA, and even EMS counterparts.

The small size of the stochastic message brings two benefits. Firstly, it reduces the hardware complexity of the non-binary decoders both in the number of interconnection wires and in the processing units themselves [20]. Secondly, shorter messages improve the average throughput when the received vector is close to a codeword and few cycles are enough for convergence. With alternative approaches, even if one iteration is required, larger messages have to be passed between the nodes resulting in reduced throughput.

It is also important to point out the difference between an SPA, MSA, or EMS decoding iteration and a stochastic decoding iteration. In the former case during one iteration the nodes exchange full pmf messages, while in the stochastic case only one symbol is exchanged per edge. To emphasize this difference, the stochastic decoding iteration is referred to as a decoding cycle (DC) [33].

Finally, SPA, MSA, and EMS decoders follow a deterministic trajectory, and when a local optimum is reached it results in decoding failure. Stochastic decoders, on the other hand, follow stochastic trajectories, meaning that repeating the decoding process can yield an alternative path that avoids the local optimum. It has been shown in [22] that stochastic decoders are capable of decoding some of the frames that initially failed to decode. The method involves restarting the decoder with the same received soft-values vector but using a different random sequence. This technique is called redecoding, and was shown to improve the BER performance and lower the error floors.

Stochastic decoding was proposed in [19] for the binary case, then extended to non-binary LDPC codes in [24]. Using the same message names as in Figure 1, but noting that here a message is no longer a pmf, but rather a GF(q) symbol, the algorithm can be described as follows:

1) *Initialization*
   For each variable node v, the output messages $\{U_{vm_i}\}_{i=0,\ldots,d_v-1}$ are initialized according to the channel likelihood values obtained from the channel model.
2) *Variable node update*
   Given the variable node input messages V_{iv}, the output messages U_{vm} are computed:
   $$U_{vm} = \begin{cases} a & \text{if } V_{iv} = a \text{ for all } i \neq m \\ \xi & \text{otherwise} \end{cases} \quad (4)$$
   where ξ is a random sample generated from the U_{vm} statistics. In other words, if the input messages on all edges other than the current one agree on the same value a, this value becomes the output for the current edge. When such an agreement is not reached, a value is randomly picked from a stored history of U_{vm} values.
3) *Permutation step*
   In the direction from variable node v to the check node c, the permutation operation is given by:
   $$U_{mc} = h_{vc} U_{vm} \quad (5)$$
   where h_{vc} is the non-zero GF(q) value from check matrix H corresponding to the edge from v to c. For the inverse direction, when sending a message from a check node to a variable node, the value $h_{cv} = h_{vc}^{-1}$ is used.
4) *Check node computation*
   A non-binary check node output message V_{cm} is computed by summing the input messages from all edges except the edge corresponding to the output message itself:
   $$V_{cm} = \sum_{i=1, i\neq m}^{d_c} U_{ic} \quad (6)$$
   where the sum is over GF(q).

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

A common problem for stochastic decoders using the above algorithm is latching, the undesired scenario when a group of nodes form a cycle and lock into a state of reduced or no switching resulting in poor bit-error-rate (BER) performance [33], [21]. It has been shown that such non-binary stochastic decoders are only functional for GF(q) when $q \leq 16$ [24].

C. Relaxed Half-Stochastic Decoding Over GF(q)

The Relaxed Half-Stochastic (RHS) decoding algorithm for LDPC codes was proposed in [22] and represents a combination of SPA and stochastic decoding techniques. The algorithm uses successive relaxation to convert stochastic streams into LLR values. An RHS decoder can be seen as a hybrid decoder operating in both LLR and stochastic domains. Structurally, the difference between an RHS and a stochastic decoder is the variable node, with the interleaver and the check nodes being identical.

In the non-binary version, Tracking Forecast Memories (TFMs) [34] are used to store the probabilities associated with the corresponding stochastic streams. For an incoming symbol $x \in GF(q)$ the memories are updated according to the following rule:

$$\text{PMF}_t[i] = \begin{cases} (1-\beta)\text{PMF}_{t-1}[i] + \beta & \text{if } i = x \\ (1-\beta)\text{PMF}_{t-1}[i] & \text{otherwise} \end{cases} \quad (7)$$

for all $i \in GF(q)$ where $\text{PMF}_t[i]$ is the probability of x being equal to symbol i at time t, and $\beta \leq 1$ is the relaxation paramter.

The RHS algorithm was successfully generalized for GF(q) and was shown in [23] to have a performance close to that of SPA. Additionally, in [35] an optimized version of RHS was introduced for the case when variable nodes are of degree two, called RD2, that eliminates the need to perform term-by-term pmf multiplications in the variable node. In both cases, the computational complexities are lower than previously reported for non-binary LDPC decoders, but a fully-parallel implementation still remains challenging.

D. Non-binary LDPC Decoders in Literature

Table I shows a few non-binary decoders reported in literature. In [26] Spagnol et al. propose a serial architecture and FPGA implementation for a GF(8) LDPC decoder. In [25] a partially-parallel implementation of the EMS algorithm is given for GF(4). In [27] an architecture for decoding non-binary quasi-cyclic LDPC codes is proposed along with an ASIC implementation for a (620, 310) GF(32) code. An architecture that works for higher order fields GF($q \geq 64$) is presented without implementation in [17]. An optimized version of this architecture with an FPGA implementation is provided in [18].

More recently, several partially-parallel implementations have been reported [36], [37], [38] for codes over GF(32), with clock frequencies in the 200-260 MHz range, and reported throughputs in the range of 47-66 Mbit/s.

A fully-parallel implementation for a (160,80) code over GF(64) is reported in [28], and the details are included for comparison in Table I.

TABLE II
EQUIVALENCE OF OPERATIONS ON A PROBABILITY MASS FUNCTION AND A MULTISET REPRESENTATION

	pmf	multiset representation
Increase probability	$p_\alpha \leftarrow p_\alpha + \Delta_1$	$S \leftarrow S \cup \{k_1 \text{ inst. of } \alpha\}$
Normalization	$p_i \leftarrow \frac{p_i}{\sum_{j=1}^{q} p_j}$ for all i	Not needed. S is implicitly normalized.
Sampling	$c_i \leftarrow \sum_{j=1}^{i} p_j$ $r \leftarrow U(0,1)$ $s \leftarrow \underset{i}{argmin}\lvert c_i - r\rvert$	$s \leftarrow$ rand. symb. from S

III. The Adaptive Multiset Stochastic Algorithm

In this section we propose a new stochastic decoding algorithm called the Adaptive Multiset Stochastic Algorithm (AMSA). The prominent feature of this algorithm is that it scales gracefully with the order of the field q both in terms of run-time and hardware complexity, making it possible to implement practical fully-parallel LDPC decoders over higher order fields like GF(64) or GF(256).

AMSA relies on the properties of multisets, a generalization of the concept of sets that allow for multiple instances of the same element, for efficient storage and computation of beliefs, but also on the inherent simplicity of degree-two variable nodes.

The algorithm enables a substantially less complex variable node, while retaining the already simple permutation nodes and check nodes as in [15] and [24], respectively.

In what follows we show how non-binary stochastic LDPC decoding can be expressed in terms of operations on multisets, define AMSA, and give its computational complexity.

A. Multiset Representation of a Probability Mass Function

Multisets, a generalization of sets, allow for multiple instances of the same element. The cardinality of a multiset S is denoted by $|S|$ and represents the total number of instances of elements.

Definition Let S be a multiset containing symbols from GF(q) with $f_j = \frac{n_j}{|S|}$ being the probability of finding symbol α_j in S, where n_j is the number of times α_j appears in S. A random variable s that takes on values in GF(q) has an associated pmf which is defined by the probabilities $p_j = \Pr(s = \alpha_j)$.

As seen in the example provided in Section II-B a pmf can be computed from any multiset of GF(q) symbols. Here we show that it is always possible to construct a multiset representation of a given pmf, such that the difference between the probabilities p_j and f_j is less than ε, for any $\varepsilon > 0$, and all $1 \leq j \leq q$.

Proposition 3.1: For any given pmf defined by probabilities p_j and any given $\varepsilon > 0$, there exists a multiset S such that $\left| p_j - \frac{n_j}{|S|} \right| < \varepsilon$ for all $1 \leq j \leq q$. (See Appendix A for the proof.)

In fact, a multiset can be used as an approximate representation of a probability mass function that has the advantage of being simple to sample and not requiring normalization.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

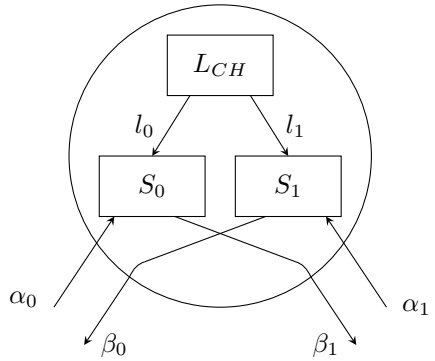

Fig. 2. Schematic representation of a variable node of degree two with the multisets S_0 and S_1, and the channel likelihood table L_{CH}. The incoming messages are α_0 and α_1, and the outgoing messages are β_0 and β_1.

Table II illustrates that in order to increase a probability p_α by Δ_1, one or more instances of symbol α are added to S. The exact number of instances to add, k_1, can be calculated by solving the equation $\frac{n_\alpha}{|S|} + \Delta_1 = \frac{n_\alpha + k_1}{|S|+k_1}$. Sampling a pmf involves calculating a cumulative density function (CDF) c_i, and finding i where c_i is closest in value to r, the realization of a uniform random variable between 0 and 1. In contrast, sampling a multiset representation of a pmf is trivial, and is equivalent to picking a random symbol from S.

B. Algorithm Definition and Analysis

In [23], [35] pmfs are used to represent the statistics of the stochastic streams associated with edges in the Tanner graph. However, the variable node update involves recalculating q probabilities, and sampling takes up to q steps and requires computing or updating a cumulative density function.

AMSA addresses these problems by using multisets instead of pmfs. Figure 2 illustrates the structure of a degree two variable node with its input symbols α_0 and α_1, which, together with the corresponding channel likelihoods l_0 and l_1, are used to update the multisets S_0 and S_1. These multisets are associated with the two edges of the variable node. The output symbols are samples from these multisets.

In the context of this configuration, we define three routines that operate on multisets: the *Add* routine (Algorithm 1), the *Remove* routine (Algorithm 2), the *Sample* routine (Algorithm 3). Together these routines constitute AMSA.

1) The Add Routine: This routine is shown in Algorithm 1 and updates a multiset S by adding zero or more instances of incoming symbol α to it. When one or more symbols are added the probability p_α associated with α is increased while the probabilities of all other symbols are decreased. The routine makes use of the floor operator because only an integer number of symbols can be added to S. The fractional part is compared against r_1, the realization of a uniform random variable, to decide whether or not an additional symbol should be added. It is easy to show that the expected number of symbols added by this routine is $l_\alpha(M-|S|)$. The term $M-|S|$ is the difference between the maximum capacity of S and its current size, and can be interpreted as the empty part of S, or the spare capacity of S.

Algorithm 1: The *Add* routine. M is the maximum size of $|S|$ given by limited memory capacity, α is the incoming symbol, l_α is the channel likelihood of symbol α.

Input: S, α, l_α
Output: Updated S with zero, one, or more instances of α added

$r_1 \leftarrow$ realization of random variable uniformly distributed on $(0, 1)$
$x \leftarrow l_\alpha \cdot (M - |S|)$
if frac$(x) \geq r_1$ **then**
  $k \leftarrow \lfloor x \rfloor + 1$
else
  $k \leftarrow \lfloor x \rfloor$
end
$S \leftarrow S \cup \{$ k instances of $\alpha\}$

Now let us examine what is the effect of adding k instances of symbol α to S on p_α, the probability of α according to S, and p_β, where β is a symbol from GF(q) other than α. Before the addition: $p_\alpha(t) = \frac{n_\alpha}{|S|}$ and $p_\beta(t) = \frac{n_\beta}{|S|}$, and after addition: $p_\alpha(t+1) = \frac{n_\alpha + k}{|S|+k}$ and $p_\beta(t+1) = \frac{n_\beta}{|S|+k}$.

Expressing $p_\alpha(t+1)$ in terms of $p_\alpha(t)$:

$$\frac{p_\alpha(t+1)}{p_\alpha(t)} = \frac{n_\alpha + k}{|S|+k} \cdot \frac{|S|}{n_\alpha}$$

$$p_\alpha(t+1) = p_\alpha(t) \cdot \frac{|S|}{|S|+k} \cdot \frac{n_\alpha + k}{n_\alpha}$$

$$= p_\alpha(t) \cdot \left(1 - \frac{k}{|S|+k}\right) \cdot \left(1 + \frac{k}{n_\alpha}\right)$$

$$= p_\alpha(t) \cdot \left(1 - \frac{k}{|S|+k}\right) + p_\alpha(t) \left(1 - \frac{k}{|S|+k}\right) \frac{k}{n_\alpha}$$

$$= p_\alpha(t) \cdot \left(1 - \frac{k}{|S|+k}\right) + \frac{n_\alpha}{|S|} \frac{|S|}{|S|+k} \frac{k}{n_\alpha}$$

$$= p_\alpha(t) \cdot \left(1 - \frac{k}{|S|+k}\right) + \frac{k}{|S|+k}$$

If we substitute k by its expected value $l_\alpha(M-|S|)$ and denote $\omega_\alpha = \frac{l_\alpha(M-|S|)}{|S|+l_\alpha(M-|S|)}$, then the equation becomes:

$$p_\alpha(t+1) = p_\alpha(t) \cdot (1 - \omega_\alpha) + \omega_\alpha$$

Similarly, expressing $p_\beta(t+1)$ in terms of $p_\beta(t)$ we get:

$$p_\beta(t+1) = p_\alpha(t) \cdot (1 - \omega_\alpha)$$

Now, we can write the update equation of the *Add* routine for the incoming symbol $\alpha \in GF(q)$ as:

$$p_s(t+1) = \begin{cases} (1-\omega_\alpha) \cdot p_s(t) + \omega_\alpha & \text{if } s = \alpha \\ (1-\omega_\alpha) \cdot p_s(t) & \text{otherwise} \end{cases} \quad (8)$$

where $p_s(t)$ is the probability of symbol s at time t. Note that this update maintains the probabilities normalized, i.e. $\sum_{s \in GF(q)} p_s(t) = 1$ for all t.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

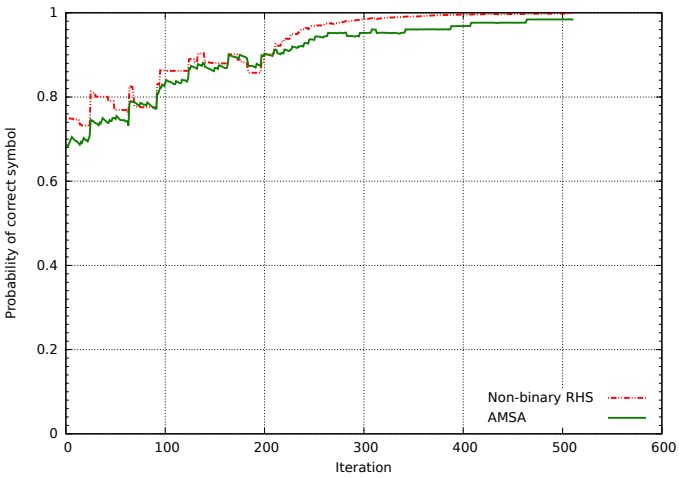

Fig. 3. Comparison of the non-binary RHS algorithm and AMSA by tracking the probability of the correct symbol in an edge memory. The code is (192,96) (2,4)-regular with $d_v = 2$. The maximum size of the multiset is $M = 128$.

Algorithm 2: The *Remove* routine of AMSA

Input: S
Output: Updated S with possibly one symbol removed

$r_2 \leftarrow$ realization of random variable uniformly distributed on $[1, M]$
if $r_2 < |S|$ **then**
  $S \leftarrow S \setminus \{\text{random symbol from } S\}$
end

Equation (8) is recognizable as the RHS update rule from (7) but instead of using a constant term β, it uses ω_α, a function of the likelihood l_α corresponding to incoming symbol α.

This result is confirmed by experimental data. Figure 3 shows how the probability of the correct symbol evolves during the decoding process in a TFM using the non-binary RHS update from (7) and AMSA as defined in this section. At any given point in time, the difference between the two probabilities is due to the fact that the multiset used by AMSA is only an approximation of a pmf.

2) The Remove Routine: The goal of the *Remove* routine is to uniformly and gradually remove symbols from multiset S. The decision whether to remove a symbol or not is based on the result of a probabilistic experiment of comparing r_2, the realization of a uniform random variable, from the interval $[1, M]$ to $|S|$. It can be shown that the expected number of symbols removed by *Remove* is $\frac{|S|}{M}$. Intuitively, that means that when $|S|$ is closer in value to M, i.e. S is close to maximum capacity, it is more likely that a symbol will be removed, and as $|S|$ approaches zero, i.e. S becomes empty, it is less likely that a symbol will be removed. Note that the *Remove* routine enforces the lower bound $1 \leq |S|$ on the cardinality of S, because r_2 is not smaller than 1.

Lemma 3.2: Let n_α be the number of times symbol α appears in S, then the probability that the Remove routine removes symbol α is $\frac{n_\alpha}{M}$.

Algorithm 3: The *Sample* routine of AMSA

Input: S
Output: A random symbol from S

$\alpha \leftarrow$ a random symbol from S

As it is shown in Proposition 3.3, *Remove* does not change the expected value of the probabilities of the symbols in S. This means that by invoking the *Add* and *Remove* routines, the expected values of the probabilities will be updated as in (8).

Proposition 3.3: Let $p_\alpha(t)$ be the probability of symbol α in S, and let $E(p_\alpha(t+1))$ be the expected probability of the same symbol after the invocation of *Remove*, then $E(p_\alpha(t+1)) = p_\alpha(t)$, i.e. the expected value of the probability is not changed by *Remove*.

Proof: Looking at a symbol α from S, the *Remove* routine is an experiment with three outcomes: no symbol is removed, symbol α is removed, and another symbol $\beta \neq \alpha$ is removed. Let p_{oi} be the probability of outcome i, then $p_{o1} = 1 - \frac{|S|}{M}$, and, from Lemma 3.2, $p_{o2} = \frac{n_\alpha}{M}$, and, finally, $p_{o3} = \frac{|S|-n_\alpha}{M}$.

When no symbol is removed, the probability is unchanged, and remains equal to $\frac{n_\alpha}{|S|}$. When α is removed, its new probability is $\frac{n_\alpha - 1}{|S|-1}$. Finally, when a symbol other than α is removed, the probability of α is $\frac{n_\alpha}{|S|-1}$. We can now compute the expected value:

$$\begin{aligned}
E(p_\alpha(t+1)) &= \left(1 - \frac{|S|}{M}\right)\frac{n_\alpha}{|S|} + \frac{n_\alpha}{M}\frac{n_\alpha - 1}{|S|-1} \\
&\quad + \frac{|S|-n_\alpha}{M}\frac{n_\alpha}{|S|-1} \\
&= \left(1 - \frac{|S|}{M}\right)\frac{n_\alpha}{|S|} + \frac{n_\alpha^2 - n_\alpha + |S|n_\alpha - n_\alpha^2}{M(|S|-1)} \\
&= \left(1 - \frac{|S|}{M}\right)\frac{n_\alpha}{|S|} + \frac{n_\alpha}{M} \\
&= \frac{n_\alpha}{M}\left(\frac{M - |S|}{|S|} + 1\right) \\
&= \frac{n_\alpha}{|S|} = p_\alpha(t)
\end{aligned}$$

■

3) The Sample Routine: The *Sample* routine generates random symbols according to the probabilities of the symbols in S. Unlike *Add* and *Remove*, this routine does not modify S. The probability that *Sample* returns symbol α is equal to p_α.

C. Non-Binary LDPC Decoding with AMSA

Having introduced the multiset representation of probability mass functions, and the mathematical properties of the *Remove*, *Add*, and *Sample* routines, we present the steps of AMSA decoding:

1) *Initialization*. When the soft-decision sequence $\mathbf{y}$ is received from the channel, the decoder front-end uses $\mathbf{y}$ to compute, for each variable node, a set of initial likelihoods l_j for each symbols in GF(q) where $j =$

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

$1, \ldots, q$. Additionally, M random samples are generated according to the likelihoods l_j and fill the multisets S_0 and S_1.

2) *Variable node update*. The variable node processing starts with the $Remove(S_i)$ routine and is followed by the $Add(S_i, \alpha_i, l_i)$ routine, where $i = 0, 1$. The variable nodes' outputs are outcomes of the $Sample$ routine from S_0 and S_1, as depicted in Figure 1. Furthermore, the variable node computes the overall belief given by $\underset{i}{argmax}\, l_i$, where l_i is the likelihood of the input symbol α_i, and $i = 0, 1$.
During the first iteration the multisets S_0 and S_1 contain M symbols. Therefore, no symbols are added. The output symbols, in this case, are generated as usual.
3) *Permutation step* follows (5).
4) *Check node computation* follows (6).
5) *Tentative decoding*. If the beliefs computed in the variable nodes satisfy all the parity equations, decoding stops. Otherwise, decoding continues until a maximum preset number of decoding cycles is reached.

Note that in non-binary stochastic LDPC decoding, the variable update is, by far, the most complex computation, with the permutation step being equivalent to a single GF(q) multiplication, and the check node requiring only a summation of d_c values over GF(q). AMSA, and its variable node update presented here, stand out from the other approaches in the literature [33], [34], [22], [23], [39] in several ways. First, it solves the problem of variable node inputs agreement, seen in (4), where as the order of the field grows $q > 16$, it becomes increasingly unlikely that the condition is satisfied. Second, AMSA avoids the hybrid approach, like in the RHS algorithm, where both pmfs and stochastic messages are used, which requires a conversion between the two, increasing hardware complexity, especially in the non-binary case. Finally, AMSA exhibits two properties: low computational complexity (as shown in Table III), and low hardware complexity (as shown in Section IV), which enabled the implementation of a fully-parallel non-binary LDPC decoder over GF(256), which, to the best of our knowledge, is the first such implementation.

D. Complexity Analysis

Table III presents the upper bounds on the number of operations needed for each stage of the proposed decoding process. Note that in the case of the *Add* routine, the complexity is more intuitively $O(d_v M)$, where d_v is the degree of the variable node, but since M scales linearly with q for all practical purposes, it was presented as $O(d_v q)$ to simplify comparison with other results in the literature.

The second column of the table gives the upper limit on the run-time of the algorithm computations using conventional memory which requires $O(k)$ steps in order to write k identical values. This approach was used for the FPGA implementation presented in Section IV. Whenever operations can be carried out in parallel *and* implemented as such on hardware, the corresponding reduction in complexity has been considered. For instance, the *Sample* routine is shown to require $O(d_v)$ operations as it is required for the d_v edges of the variable node. However, in the FPGA implementation all d_v edges are instantiated and can execute the routine in parallel in $O(1)$ time. In fact, all the multisets associated with edges are sampled in parallel in 1 clock-cycle. Similarly, the rest of the routines can be executed in parallel to reduce the run-time complexity.

The third column provides the run-time complexities when custom designed memories are available, which are capable of writing an identical value in multiple locations in only one clock-cycle. In this case, all the variable node computations have a run-time complexity of $O(1)$, which further increases the throughput of the decoder.

Table IV presents the memory space requirements for AMSA. Note that a GF(q) symbol is represented in hardware by $\log q$ bits and that the multisets are implemented as memories, while w is the number of bits used to represent probabilities.

TABLE III
THE NUMBER OF OPERATIONS REQUIRED AND RUN-TIME COMPLEXITY FOR FULLY-PARALLEL IMPLEMENTATIONS OF THE AMSA ALGORITHM

	Number of operations	Run-time complexity of fully-parallel implementation (conventional memory)	Run-time complexity of fully-parallel implementation (custom memory)
Add	$O(d_v q)$	$O(q)$	$O(1)$
Remove	$O(d_v)$	$O(1)$	$O(1)$
Sample	$O(d_v)$	$O(1)$	$O(1)$
Variable Node	$O(d_v q)$	$O(q)$	$O(1)$
Check Node	$O(d_c)$	$O(1)$	$O(1)$
Permutation	$O(1)$	$O(1)$	$O(1)$
Belief	$O(d_v)$	$O(1)$	$O(1)$

TABLE IV
SUMMARY OF SPACE COMPLEXITY FOR AMSA

	Space complexity	Comments
Memories	$O(d_v M \log q + qw)$	d_v edge memories and q w-bit likelihood memories
Variable Node	$O(\log M)$	determined by the size of memory indices
Check Node	$O(1)$	no memory required
Permutation	$O(q \log q)$	$q \times \log q$ lookup table

E. Redecoding

It has been shown in [22] that if a stochastic decoder fails to decode a codeword, it may succeed by trying to decode it again using different random sequences.

Redecoding can be seen as a tradeoff mechanism between error-rate performance and latency. A redecoding configuration has two parameters: the number of attempts r_a, and the maximum number of decoding cycles for each attempt $\max_{DC}$, and latency scales with the product of these two parameters. These parameters can be changed at run-time making the AMSA decoder suitable for variable latency application.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

TABLE V
COMPARISON OF GF(64) AMSA-128 AND GF(256) AMSA-512 FULLY-PARALLEL DECODERS

	GF(64) AMSA-128	GF(256) AMSA-512
Variable Nodes	192	192
Check Nodes	96	96
Edge Memories	384	384
Permutation Blocks	1152	1152
Length of symbols	6 bits	8 bits
Length of memory indices	7 bits	9 bits
Memory size	128×6 bits	512×8 bits
Length of Random Number Generator	32 bits	32 bits
Quantization parameter, w	12 bits	12 bits

The technique of redecoding has been successfully extended to non-binary stochastic decoding in this work in order to improve performance and lower the error floor.

IV. CIRCUIT IMPLEMENTATION

In this section we consider two fully-parallel AMSA decoder implementations: AMSA-128 over GF(64), and AMSA-512 over GF(256), where AMSA-M denotes a configuration of the decoder that uses multisets of maximum size M.

The GF(64) and GF(256) codes are (192,96) (2,4)-regular LDPC codes used in the DAVINCI project [13]. Table V presents a detailed comparison between the decoders.

The FPGA platform used for implementation is the EP4SGX230-KF40C2 chip from the Altera Stratix IV GX family. It provides, among other things, 182,400 Adaptive Lookup Tables (ALUTs), 182,400 registers, 1,235 M9K memory blocks, and 1,288 18x18-bit Digital Signal Processing (DSP) blocks.

A. Variable Node

The channel likelihood values are represented in fixed-point format using w bits. The q likelihood values are organized in a $q \times w$ bit dual-port memory, as illustrated in Figure 4. Within each variable node the *Add* and the belief computation routines make use of the likelihood values. In both cases the access is read-only. The only time the likelihood memory is written to is during the initialization phase of the algorithm.

The hardware representation for a multiset is a memory array of length M, an approach similar to the Edge Memories (EMs) that were introduced for the binary case in [33] and then extended to GF(16) in [24]. In other words, we impose an upper and lower bounds on the cardinality of the multisets $1 \leq |S_i| \leq M$, $i = 0, 1$. For practical reasons and efficient utilization of memory resources, M is chosen to be a power of 2. On the Altera Stratix IV FPGA platform used in this work, the so-called M9K memory blocks were used for this purpose. Each block has a capacity of 8192 bits with configurable dual read-write ports. Both codes used in this work have 384 edges in the Tanner graph, and therefore, use 384 M9K memory blocks.

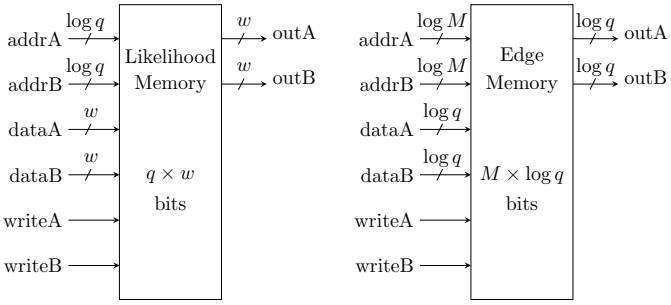

Fig. 4. The interfaces of a channel likelihood memory and an edge memory. Probabilities are are represented using w bits, and symbols using $\log q$ bits. Note that both types of memories are dual-port.

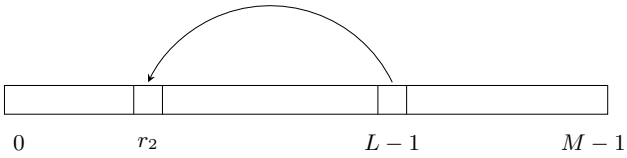

Fig. 5. Removing a symbol from memory at random index $r_2 < L$ by overwriting it with the symbol at index $L-1$ and, finally, decrementing L.

The *Add*, *Remove*, and *Sample* routines make use of randomly generated bits. This implementation uses a 32-bit linear feedback shift register (LFSR) in each variable node. The LFSRs have $x^{32} + x^{22} + x^2 + x + 1$ as feedback polynomial and achieve the maximum-length sequence of $2^{32} - 1$ [40].

1) Implementation of the Remove Routine: On the hardware implementation, we represent S with a memory of length M, which is implicitly an ordered collection. Normally, if the order of the elements had to be preserved, removing an arbitrary element from the memory would imply shifting up to $M-1$ elements by one position to the left, which would take $O(M)$ steps to perform. Fortunately, in AMSA we are not concerned with the order of the elements in the memory. In this case, removing an element can be achieved by overwriting it with the last element in the sequence and reducing the length of the sequence by one. This operation takes only $O(1)$ steps.

The hardware implementation of the *Remove* routine can be expressed as in Algorithm 4, where L is the size of the multiset, and $MEM[i]$ represents the ith element in the memory array. The $r_2 < L$ test is done by a $\log M$-bit comparator which controls the read-write mode of the edge memory. Note that it is possible to avoid explicitly fetching the value $MEM[L-1]$ from the memory each time. It can be stored and updated in a register, denoted as $LastSymbol$ in Figure 6.

2) Implementation of the Add Routine: The circuit for computing k, the number of symbols to add, is given in Figure 7, and it uses a $w \times \log M$-bit multiplier and two w-bit comparators and two multiplexers. On the FPGA system used here, each of the 1,288 DSP blocks provides a 18×18-bit multiplier, more than enough to instantiate one for each of the 384 edges. On platforms where DSP blocks are not available, it is possible to use truncated multipliers [41], [42].

When using conventional memory modules, the task of writing k symbols to the edge memory is controlled by a state machine as in Figure 8.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

Algorithm 4: The set of commands implementing the *Remove* routine.

```
1 r2 ← LFSR
2 if r2 < L then
3 |   MEM[r2] ← MEM[L − 1]
4 |   L ← L − 1
5 end
```

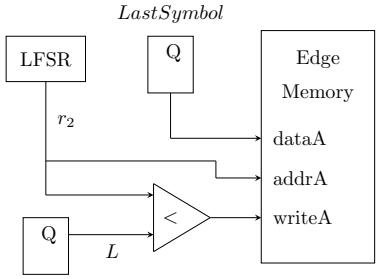

Fig. 6. Circuit for the *Remove* routine.

3) Implementation of the Sample Routine: One way to generate random integers in the $[0, L-1]$ range, when L is variable and not necessarily a power of 2, is to first generate a random integer $X \in [0, 2^p - 1]$ where $L \leq 2^p$ and then compute the remainder of X/L. Unfortunately, using an integer divider is not practical. Alternatively, we use an adapted version of the acceptance-rejection sampling method [43]. Since the *Sample* routine has to generate a sample at every invocation, we use a series of fallback values in case of rejection as shown in Figure 9. Here r_3 is a realization of a uniform random variable, and $r_3(3:7)$ represents the sequence of bits $b_3, \ldots, b_7$ from r_3 where b_1 is the most significant bit, and $L(1:2)r_3(3:7)$ stands for the concatenation of the specified sequences of bits.

Three $\log M$-bit comparators are used in parallel to implement three acceptance-rejection tests. If the sample used in the first test passes the test it is routed to the output x, otherwise we fallback to the sample on the next level, and if necessary next level, and so on. If all the tests fail, the last fallback value is $L-1$ which is guaranteed to be a valid sample. One can create a longer chain of comparators for more uniform sampling however, the tradeoff is the longer critical path for the computation of x.

Observe that the final distribution of values of x is not truly uniform in the mathematical sense, but rather biased towards values closer in value to L, this is due to replacing the most significant bits in R with those from L. This bias can be directed to the other end of the $[0, L-1]$ interval by replacing the most significant bits of R with zeros. Alternatively, the bias can be reduced by using a random bit and multiplexing among the two options. Experimental results have shown that AMSA decoders converge faster to the correct codeword when sampling is biased like in the former case (as shown in Figure 9) even if compared with uniform sampling. This is due to the fact that the *Add* routine adds new incoming symbols at the end of the memory, resulting in a non-uniform ordering of the symbols, and in this case using more recent results seems more suitable. An alternative approach is to modify the *Add* routine to randomly interleave the symbols as it adds them to the memory, resulting in a slight improvement in performance, but at the cost of increased hardware complexity and additional memory operations.

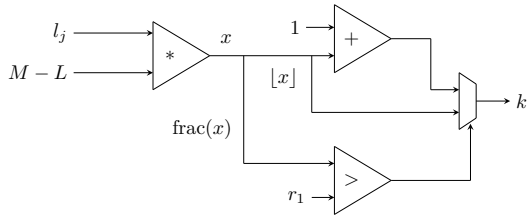

Fig. 7. A circuit for the part of the *Add* routine responsible for computing the number of symbols to add. The result of the product x is $w + \log M$ bits long, with frac(x) corresponding to the w least significant bits and the rest corresponding to $\lfloor x \rfloor$.

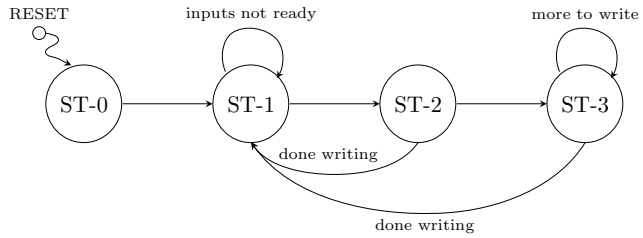

Fig. 8. The finite state machine controlling the computation in the variable node in the case of the FPGA implementation. State ST-0 is active only during initialization, it loads the q likelihood values into the likelihood memory and the M initial symbols into the edge memories. State ST-1 serves as a synchronization point, while memories are written during states ST-2 and ST-3. For the detailed memory access schedule see Appendix B.

B. Check Node

The check node in stochastic decoders is considerably simpler than the SPA equivalent, as seen in (6), and it is a sum over GF(q), implementable directly with XOR gates. Besides the output messages, the check nodes have an additional output bit to signal whether the parity check is satisfied, which is determined based on the belief symbols of the variable nodes, rather than the edge outputs.

C. Permutation Block

On each edge of the Tanner graph (see Figure 10) there are three GF(q) multiplication-by-constant operations performed: one for the message from the variable node to the check node, another one for the reverse direction, and one for the belief message from the variable node to the check node. Even though the number of such blocks is large, as shown in Table V, the hardware complexity of each of them is small.

A GF(q) multiplication by a constant is efficiently implemented by a $q \times \log q$-bit LUT, assuming q is a power of 2. Modern FPGA platforms provide 6-input LUT resources which can be used directly for $q \leq 64$ or combined for $q > 64$. A GF(64) permutation nodes is implemented using six 6-input LUTs while the GF(256) version requires fifteen 6-input LUTs.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

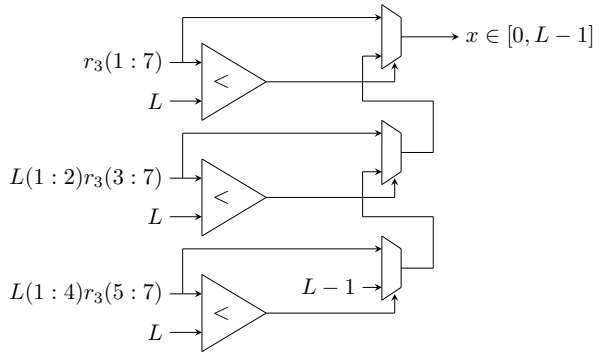

Fig. 9. A circuit for the *Sample* routine as implemented for AMSA-128, where the output x is used as address on the edge memory in order to read a random symbol.

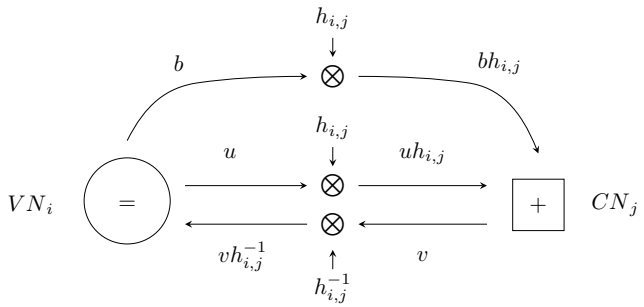

Fig. 10. The edge of the Tanner graph connecting variable node i (VN_i) and check node j (CN_j). The $h_{i,j}$ multiplication factor is the non-zero value from H matrix associated with this edge. Message b is the variable node belief.

D. Synthesis Results

The FPGA synthesis results for the GF(64) AMSA-128 and GF(256) AMSA-512 designs are given in Table VI, while Figure 11 shows the floor plan of the FPGA chip for the GF(256) AMSA-512 decoder.

Synthesis results confirm the complexity analysis done in Section III-D and summarized in Table IV. The total amount of memory used scaled, as predicted, with $O(d_v M \log q + qw)$.

Similarly, the size of the variable node control logic, which is built using ALUTs and registers, scales, as predicted, with $O(\log q)$.

Note that the number of memory blocks did not increase because in both cases the number of edge memories is the same, and each edge memory fits in a M9K block.

E. ASIC-specific Considerations

In this section, we combine the *Add* and *Remove* routines into a single memory write operation, based on the observation that since both routines are non-deterministic, there are three possible scenarios to consider (see Figure 12). We also note that by designing a memory with a custom address decoder that enables multiple cells at the same time this write operation can be done in one cycle.

Figure 13 provides an architecture for such an SRAM memory. The Address Decoder used in the circuit is a standard address decoder. The Mask Overlay unit sets high all the address lines in the segment $[L, M-1]$. Thus, having selected multiple SRAM cells, the data will be written to multiple locations of the memory in one cycle. Even though more than the necessary k symbols are written (see Algorithm 1 line 8) in the $[L, M-1]$ segment of the memory, only k will be in S, the rest being outside and, thus, not having any effect.

We can consider the Address Decoder and Mask Overlay pair as a Custom Address Decoder for the SRAM memory block. In order to estimate how much bigger is the Custom Address Decoder compared to the standard Address Decoder, both have been implemented in VHDL and compared in terms of logical resources used. In the case of $M = 128$ the size increased by 43% while for $M = 256$ the size increased by 38%. Considering that the custom address decoder represents only a part of the total area of an SRAM block (the rest being occupied by the SRAM cells, sense amplifiers, etc.), the overall area increase for an SRAM memory block is rather insignificant.

The use of such custom SRAM memory enables an ASIC-specific architecture of the decoder, which at the cost of additional memory controller complexity increases throughput significantly (as shown in Section V-B).

TABLE VI
SUMMARY OF THE HARDWARE RESOURCES USED BY THE FULLY-PARALLEL GF(64) AMSA-128 AND GF(256) AMSA-512 FULLY-PARALLEL DECODERS ON ALTERA STRATIX IV GX EP4SGX230. NOTE THAT THESE RESULTS ARE AFTER PLACE AND ROUTE.

Resource	Total available on FPGA	GF(64) AMSA-128 (fully-parallel)	GF(256) AMSA-512 (fully-parallel)
Adaptive Look-up Tables	182,400	66,885 (37%)	91,376 (50%)
Adaptive Logic Modules	91,200	0	0
Registers	182,400	23,150 (13%)	30,840 (17%)
Simple multipliers (12x12)	1,288	384 (30%)	384 (30%)
M9K memory blocks	1,235	576 (47%)	576 (47%)
Total block memory bits	14,625,792	453,120 (3%)	2,162,688 (15%)

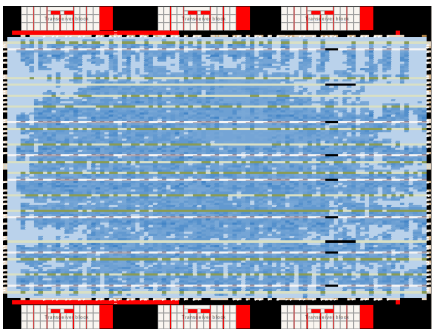

Fig. 11. The FPGA chip floor plan after the synthesis, and place and route of the GF(256) AMSA-512 fully-parallel decoder. Different colors indicate different type of hardware resources used, while darker shades indicate higher percentage of usage of the logic elements in a particular block.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

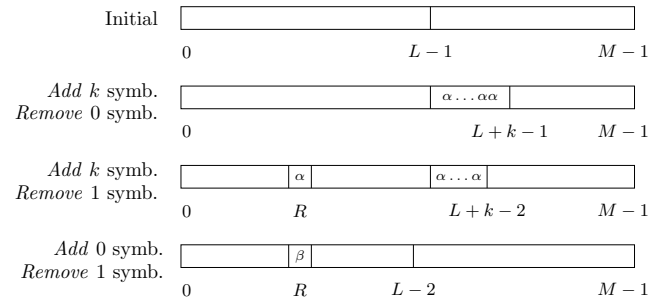

Fig. 12. Schematic representation of the possible scenarios based on the outcomes of the *Add* and *Remove* routines and the changes made to the memory.

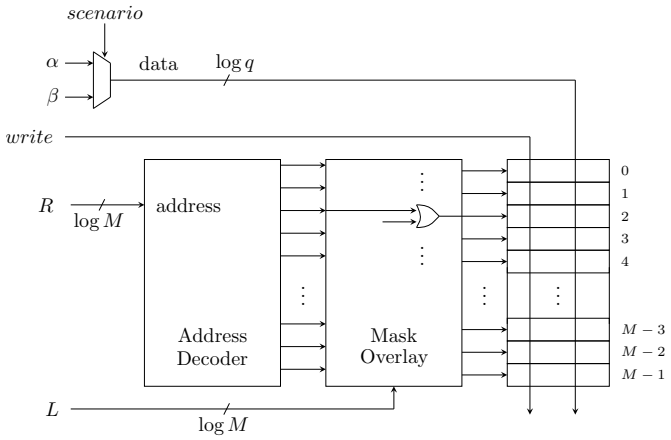

Fig. 13. Architecture of full-custom memory module that can perform any of the scenarios in Figure 12 in one cycle.

V. Simulation Results and Analysis

A. Performance

The performance of several configurations of the AMSA algorithm are given in Figure 14 and Figure 15 for the Additive White Gaussian Noise (AWGN) channel. As it can be seen, the decoder configurations with larger values of M have better performance. This is because when M is larger the multiset representation is more precise, as it was shown in Section III-A.

Another parameter that affects performance is redecoding. In Figure 14 and Figure 15, the performance is compared for the same configuration with and without redecoding but keeping the total number of decoding cycles equal. For example, for the GF(64) AMSA-256 decoder the frame error rate at $E_b/N_0 = 2.4$dB is 6×10^{-6} with 5×10^4 maximum allowed decoding cycles, while if doing 5 redecoding attempts of 10^4 maximum decoding cycles each, the performance is improved to 3×10^{-6}.

For the GF(64) code, AMSA-512 is only 0.04 dB away from the floating-point SPA performance at an FER of 2×10^{-6}. On the other hand, when using the GF(256) version of code, the difference between the SPA results and AMSA-1024 are of about 0.2 dB at an FER of 4×10^{-6}. Note that the SPA algorithm used here uses a floating-point representation, and a fully-parallel SPA decoder for the codes used in this work is, at the moment, impractical.

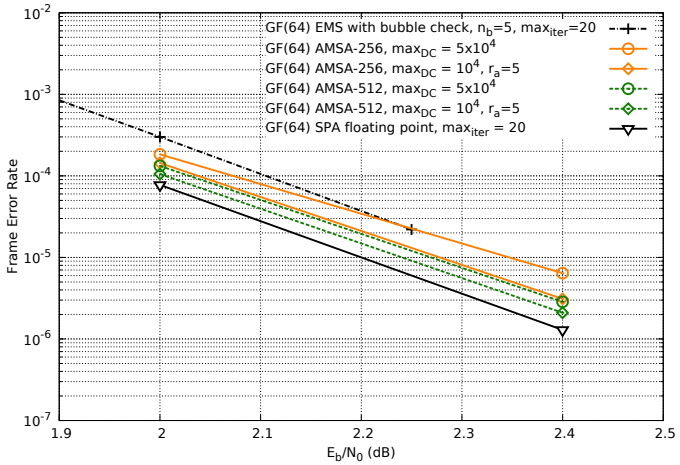

Fig. 14. Frame error rate performance of AMSA compared to the Bubble Check algorithm from [18] and SPA. The code used here is (192,96) (2,4)-regular over GF(64). Parameter r_a stands for the number of redecoding attempts. For all AMSA configurations likelihoods are represented in $w = 12$ bits, EMS uses a log-domain representation using 5 bits, and SPA uses a floating point representation.

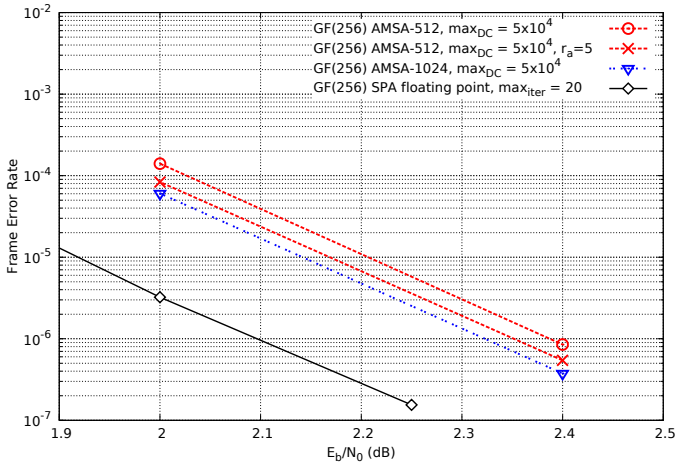

Fig. 15. Frame error rate performance of AMSA compared to SPA. The code used here is (192,96) (2,4)-regular over GF(256). The rest of the parameters are the same as in Figure 14.

B. Throughput and Latency

As the SNR increases, the AMSA decoder takes fewer decoding cycles to complete decoding. This implies that the throughput is a function of the SNR, as shown in Figure 16 where in order to simplify comparison of throughput of the conventional memory and custom memory architectures, we use the same clock frequency in both cases.

As Table I shows, the highest reported throughput for a hardware implementation for the GF(64) (192,96) code is 3.8 Mbit/s at an SNR of 2.4 dB [18] using EMS with the Bubble Check algorithm. We note that, in this case, AMSA presents a significant improvement in throughput.

It is interesting to compare the performance results of this implementation to the one in [28], which has a reported throughput of 1.15 Gbit/s and uses a code over GF(64). We note that AMSA-256 GF(64) implementation achieves

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

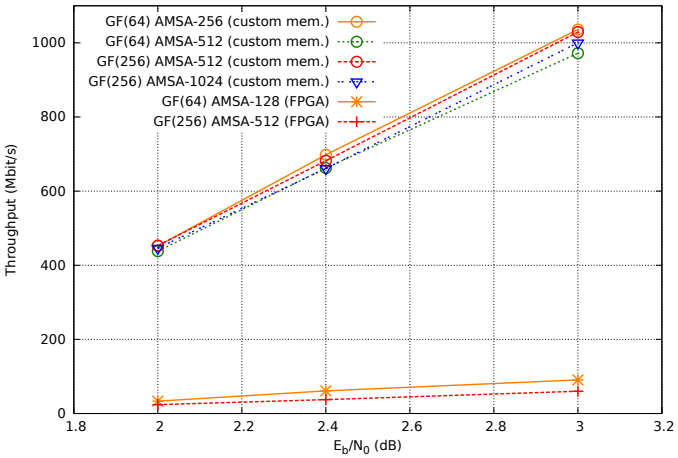

Fig. 16. Coded throughput of the FPGA implementation that uses conventional memory, and estimated throughput for an AMSA implementation that uses the custom memory architecture, based on the GF(64) and GF(256) version of the (192,96) (2,4)-regular DAVINCI code, at clock frequency f = 108 MHz.

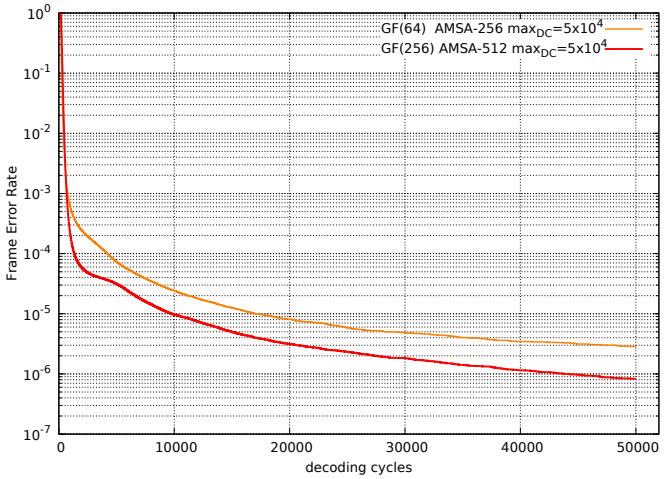

Fig. 17. Settling curves for the GF(64) and GF(256) versions of the (192,96) code at an SNR of 2.4 dB.

the same throughput at an SNR of approximately 3.1 dB, even though the clock frequency in this case is much lower. Additionally, the error correction performance in [28] is good in the $3.2-3.6$ dB SNR range, but very limited in the $2.0-2.4$ dB range used in this work.

In this case the limit on the number of decoding cycles max_{DC} is set to 5×10^4 even though the average number of decoding cycles at E_b/N_0 of 2.4 dB is approximately 180. Note that while there are multiple possible redecoding configurations for a given total number of decoding cycles, some of these configurations perform better than the others. In the case of AMSA-256 GF(64) decoder, the best results seem to be obtained with configurations like $r_a = 5, \max_{DC} = 10^4$, or $r_a = 10, \max_{DC} = 5 \times 10^3$, while configurations where $\max_{DC} < 5 \times 10^3$ generally perform poorly. This observation is confirmed by the settling curves plot presented in Figure 17, which shows the frame error rate settle down slowly as the number of decoding cycles increases.

For the $O(1)$ run-time complexity ASIC architecture a decoding cycle corresponds to one clock cycle, meaning that for a clock frequency of $f = 108$MHz, the latency introduced by a decoder with $max_{DC} = 5 \times 10^4$ is approximately 0.5 ms.

In the case of SPA, the average number of iterations at E_b/N_0 of 2.4 dB is approximately 4 for both the GF(64) and GF(256) version of the code. In this case latency cannot be directly estimated without knowing how many clock cycles an SPA iteration takes for the particular implementation, though a fully-parallel implementation of non-binary SPA decoders for these codes seems to be impractical.

VI. Conclusion

Based on a multiset representation for probability mass functions, the Adaptive Multiset Stochastic Algorithm was introduced and applied for the non-binary stochastic decoding of LDPC codes with $d_v = 2$. Additionally, the concept of redecoding was applied to non-binary LDPC decoding and shown to improve the decoding performance.

AMSA was used for the FPGA implementation of two fully-parallel LDPC decoders over GF(64) and GF(256). To the best of our knowledge, this is the first reported fully-parallel LDPC decoders over GF(256).

The GF(64) decoder achieves a coded throughput of 65 Mbit/s at $E_b/N_0 = 2.4$ dB when using conventional memory, and 698 Mbit/s at the same E_b/N_0 when using custom memory. At an FER of 2×10^6 the GF(64) version is only 0.04 dB away from the floating-point SPA performance, while the difference for the GF(256) decoders is of 0.2 dB. Furthermore, these implementations demonstrate the highest throughput reported for the particular codes used.

A memory architecture was proposed that reduces the run-time complexity of an AMSA decoding cycle to $O(1)$. The estimated throughput for a fully-parallel decoder using this architecture is of 698 Mbit/s for the GF(64) decoder, and 512 Mbit/s for the GF(256) decoder at 2.4 dB and at a clock frequency of 108 MHz.

There are several possible direction for future work like the generalization of the algorithm for $d_v > 2$ (especially if better $d_v > 2$ non-binary high-order fields codes become available), gaining better understanding of the redecoding process and how to optimally choose the r_a parameter, evaluating the impact on circuit power consumption from employing the presented decoding techniques, understanding why the decoding performance gap between SPA and AMSA is larger for GF(256) than for GF(64), improving the latency, etc.

Appendix A
Proof of Proposition 3.1

Proof: Let $n_j = \text{nint}(|S|p_j)$, where nint is the nearest integer function also known as the round function. Then

$$\frac{n_j}{|S|} = \frac{\text{nint}(|S|p_j)}{|S|} = \frac{|S|p_j + e(|S|p_j)}{|S|} = p_j + \frac{e(|S|p_j)}{|S|}$$

where $e(x)$ is the rounding error function for x.

The condition $\left| p_j - \frac{n_j}{|S|} \right| < \epsilon$ becomes $\left| \frac{e(|S|p_j)}{|S|} \right| < \epsilon$.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

Note that $-\frac{1}{2} < e(x) < \frac{1}{2}$ for all x, and that it is always possible to create a large enough multiset S such that

$$\frac{e(|S|p_j)}{|S|} < \varepsilon.$$

■

APPENDIX B
MEMORY ACCESS SCHEDULE FOR FPGA IMPLEMENTATION

As it is shown in Figure 2, each variable node uses three dual-port memory blocks: one to store the channel likelihoods table L_{CH}, and two to store the multisets S_0 and S_1, respectively. The following table details the memory access schedule for the state machine in Figure 8.

TABLE VII
COMPUTATIONS CORRESPONDING TO EACH STATE

State	Routine	Memory port	Action
ST-0	*Load*	memL_{CH}-A	load likelihoods
		memL_{CH}-B	load likelihoods
		memS_0-A	load symbols for edge 0
		memS_0-B	load symbols for edge 0
		memS_1-A	load symbols for edge 1
		memS_1-B	load symbols for edge 1
ST-1		memL_{CH}-A	get likelihood for input 0
		memL_{CH}-B	get likelihood for input 1
	Sample	memS_0-B	sample edge memory 0
		memS_1-B	sample edge memory 1
	Remove	memS_0-A	remove symbol from edge 0
		memS_1-A	remove symbol from edge 1
ST-2		memL_{CH}-A	–
ST-3		memL_{CH}-B	–
	Add	memS_0-A	add symbol to edge memory 0
		memS_0-B	add symbol to edge memory 0
		memS_1-A	add symbol to edge memory 1
		memS_1-B	add symbol to edge memory 1

Alexandru Ciobanu was born in Chișinău, Moldova in 1983. He studied at the Technical University of Cluj-Napoca, Romania in the Faculty of Automation and Computer Science before transferring to McGill University, Montréal, Québec, Canada where he recieved the B.Eng. degree in computer engineering in 2008, and the M.Eng. degree in electrical engineering in 2012.

From 2006 to 2007 he was a Software Developer at Matrox Electronic Systems Ltd. where he worked on a library of high-performance machine vision and image analysis algorithms. Between 2008 and 2011, he was a Software Developer at IBM Canada Ltd. working on data analytics software. Since 2011, he is a Software Developer at Rogue Reseach Inc. working on imaging, communication, and control software. His research interests include low-complexity error-control coding, design and implementation of iterative decoders, and digital signal processing systems.

Saied Hemati (S'01-M'06-SM'08) received the B.Sc. and the M.Sc. degrees in Electronics Engineering from Isfahan University of Technology, Isfahan, Iran, in 1993 and 1997, respectively, and the Ph.D. degree in Electrical Engineering from Carleton University, Ottawa, ON, Canada, in 2005.

From 1997 to 1999, he worked at the Integrated Electronics Industry (IEI), Isfahan, Iran. From 1999 to 2001, he was a researcher at the Research and Education Center of Isfahan Telecommunication Company, Isfahan, Iran. From 2005 to 2007, he was an NSERC postdoctoral fellow and a part-time professor at the School of Information Technology and Engineering at the University of Ottawa, Ottawa, ON, Canada. He was an assistant professor at the Department of Electrical Engineering at Sharif University of Technology, Tehran, Iran, in 2008. He was with the Department of Electrical and Computer Engineering at McGill University, Montreal, QC, Canada as a senior researcher and lecturer from 2008 to 2012. He has been an Associate Professor at the Electronics Systems division of the Electrical Engineering Department at Linköping University, Linköping, Sweden since January 2013. His research interests include the theory of operation and the design and implementation of iterative error-correcting decoders.

Dr. Hemati is the recipient of several awards and scholarships, including the Senate Medal for outstanding academic achievement, a ReSMiQ (Le Regroupement Stratégique en Microsystèmes du Québec) postdoctoral fellowship, an NSERC (Natural Sciences and Engineering Research Council of Canada) postdoctoral fellowship, a CSA (Canadian Space Agency) NSERC postdoctoral supplement award, a CITO (Communications and Information Technology Ontario) research excellence award, an Ontario Graduate Scholarship, an Ontario Graduate Scholarship in Science and Technology, and the best paper award in the CITO Knowledge Network Conference in 2002.

Warren Gross (S'92-M'04-SM'10) received the B.A.Sc. degree in electrical engineering from the University of Waterloo, Waterloo, Ontario, Canada, in 1996, and the M.A.Sc. and Ph.D. degrees from the University of Toronto, Toronto, Ontario, Canada, in 1999 and 2003, respectively. Currently, he is an Associate Professor with the Department of Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada. His research interests are in the design and implementation of signal processing systems and custom computer architectures.

Dr. Gross serves as an Associate Editor for the IEEE TRANSACTIONS ON SIGNAL PROCESSING. He is currently Vice-Chair of the IEEE Signal Processing Society Technical Committee on Design and Implementation of Signal Processing Systems. He served as Technical Program Co-Chair of the IEEE Workshop on Signal Processing Systems (SiPS 2012) and as Chair of the IEEE ICC 2012 Workshop on Emerging Data Storage Technologies. Dr. Gross is a Senior Member of the IEEE and a licensed Professional Engineer in the Province of Ontario.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TSP.2013.2264813

REFERENCES

[1] R. Gallager, "Low-density parity-check codes," *IEEE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, Jan 1962.

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

[3] S.-Y. Chung, J. Forney, G. D., T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Commun. Lett.*, vol. 5, no. 2, pp. 58–60, 2001.

[4] 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.

[5] A. Morello and V. Mignone, "DVB-S2: The second generation standard for satellite broad-band services," *Proc. IEEE*, vol. 94, no. 1, pp. 210–227, 2006.

[6] V. Oksman and S. Galli, "G.hn: The new ITU-T home networking standard," *IEEE Commun. Mag.*, vol. 47, no. 10, pp. 138–145, 2009.

[7] *IEEE Standard for Information technology-Specific requirements - Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications*, IEEE Std. IEEE 802.3an.

[8] *IEEE Standard for Information technology–Telecommunications and information exchange between systems–Local and metropolitan area networks–Specific requirements Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput*, IEEE Std. IEEE 802.11an-2009.

[9] *IEEE Standard for Local and metropolitan area networks Part 16: Air Interface for Broadband Wireless Access Systems*, IEEE Std. IEEE 802.16e.

[10] J. Lu and J. M. F. Moura, "Structured LDPC codes for high-density recording: large girth and low error floor," *IEEE Trans. Magn.*, vol. 42, no. 2, pp. 208–213, 2006.

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

[12] B. Zhou, J. Kang, S. Song, S. Lin, K. Abdel-Ghaffar, and M. Xu, "Construction of non-binary quasi-cyclic LDPC codes by arrays and array dispersions," *IEEE Trans. Commun.*, vol. 57, no. 6, pp. 1652–1662, 2009.

[13] I. Gutierrez, G. Bacci, J. Bas, A. Bourdoux, H. Gierszal, A. Mourad, and S. Pleftschinger, "DAVINCI non-binary LDPC codes: Performance and complexity assessment," in *Proc. Future Network and Mobile Summit*, 2010, pp. 1–8.

[14] A. Mourad and I. Gutierrez, "System level evaluation of DAVINCI non-binary LDPC codes," in *Proc. Future Network and Mobile Summit*, 2010, pp. 1–9.

[15] 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.

[16] X.-Y. Hu, E. Eleftheriou, D.-M. Arnold, and A. Dholakia, "Efficient implementations of the sum-product algorithm for decoding LDPC codes," in *Proc. IEEE Global Telecommunications Conf. GLOBECOM '01*, vol. 2, 2001, p. 1036.

[17] A. Voicila, D. Declercq, F. Verdier, M. Fossorier, and P. Urard, "Low-complexity decoding for non-binary LDPC codes in high order fields," *IEEE Trans. Commun.*, vol. 58, no. 5, pp. 1365–1375, 2010.

[18] E. Boutillon and L. Conde-Canencia, "Simplified check node processing in nonbinary LDPC decoders," in *Proc. 6th Int Turbo Codes and Iterative Information Processing (ISTC) Symp*, 2010, pp. 201–205.

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

[20] 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, 2008.

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

[22] F. Leduc-Primeau, S. Hemati, W. J. Gross, and S. Mannor, "A relaxed half-stochastic iterative decoder for LDPC codes," in *Proc. IEEE Global Telecommunications Conf. GLOBECOM 2009*, 2009, pp. 1–6.

[23] G. Sarkis, S. Hemati, S. Mannor, and W. J. Gross, "Relaxed half-stochastic decoding of LDPC codes over GF(q)," in *Proc. 48th Annual Allerton Conf. Communication, Control, and Computing (Allerton)*, 2010, pp. 36–41.

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

[25] Y. Sun, Y. Zhang, J. Hu, and Z. Zhang, "FPGA implementation of nonbinary quasi-cyclic LDPC decoder based on EMS algorithm," in *Proc. Int. Conf. Communications, Circuits and Systems ICCCAS 2009*, 2009, pp. 1061–1065.

[26] C. Spagnol, E. M. Popovici, and W. P. Marnane, "Hardware implementation of $GF(2^m)$ LDPC decoders," *IEEE Trans. Circuits Syst. I*, vol. 56, no. 12, pp. 2609–2620, 2009.

[27] J. Lin, J. Sha, Z. Wang, and L. Li, "Efficient decoder design for nonbinary quasicyclic LDPC codes," *IEEE Trans. Circuits Syst. I*, vol. 57, no. 5, pp. 1071–1082, 2010.

[28] Y. S. Park, Y. Tao, and Z. Zhang, "A 1.15gb/s fully parallel nonbinary LDPC decoder with fine-grained dynamic clock gating," *IEEE Int. Solid-State Circuits Conf. (ISSCC)*, 2013.

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

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

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

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

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

[34] S. Sharifi Tehrani, A. Naderi, G.-A. Kamendje, S. Hemati, S. Mannor, and W. J. Gross, "Majority-based tracking forecast memories for stochastic LDPC decoding," *IEEE Trans. Signal Process.*, vol. 58, no. 9, pp. 4883–4896, 2010.

[35] G. Sarkis and W. J. Gross, "Efficient stochastic decoding of non-binary LDPC codes with degree-two variable nodes," in *IEEE Communication Letters*, 2011, (Accepted for publication in IEEE Communications Letters).

[36] F. Cai and X. Zhang, "Relaxed min-max decoder architectures for nonbinary low-density parity-check codes," *IEEE Trans. VLSI Syst.*, to be published, early Access.

[37] X. Chen and C.-L. Wang, "High-throughput efficient non-binary LDPC decoder based on the simplified min-sum algorithm," *IEEE Trans. Circuits Syst. I*, vol. 59, no. 11, pp. 2784–2794, 2012.

[38] Y.-L. Ueng, C.-Y. Leong, C.-J. Yang, C.-C. Cheng, K.-H. Liao, and S.-W. Chen, "An efficient layered decoding architecture for nonbinary QC-LDPC codes," *IEEE Trans. Circuits Syst. I*, vol. 59, no. 2, pp. 385–398, 2012.

[39] G. Sarkis, S. Hemati, S. Mannor, and W. J. Gross, "Stochastic decoding of LDPC codes over GF(q)," *IEEE Trans. Commun.*, to be published, early Access.

[40] P. Alfke. (1996, July) Application note: Efficient shift registers, LFSR counters, and long pseudo-random sequence generators. Xilinx Inc. [Online]. Available: http://xilinx.com/support/documentation/application_notes/xapp052.pdf

[41] M. J. Schulte and J. Swartzlander, E. E., "Truncated multiplication with correction constant for DSP," in *Proc. Workshop VLSI Signal Processing, VI*, 1993, pp. 388–396.

[42] S. S. Kidambi, F. El-Guibaly, and A. Antoniou, "Area-efficient multipliers for digital signal processing applications," *IEEE Trans. Circuits Syst. II*, vol. 43, no. 2, pp. 90–95, 1996.

[43] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods (Springer Texts in Statistics)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005.

Copyright (c) 2013 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.