On-the-fly Syndrome Check for LDPC Decoders

Erick Amador and Raymond Knopp
EURECOM
06904 Sophia Antipolis, France
name.surname@eurecom.fr

Renaud Pacalet
TELECOM ParisTech
06904 Sophia Antipolis, France
renaud.pacalet@telecom-paristech.fr

Vincent Rezard
Infineon Technologies France
06560 Sophia Antipolis, France
vincent.rezard@infineon.com

***Abstract*—Modern VLSI decoders for low-density parity-check (LDPC) codes require high throughput performance while achieving high energy efficiency on the smallest possible footprint. In this paper we present a valuable optimization to the processing step known as *syndrome check*. After each decoding iteration the updated posterior values are used to verify the validity of the codeblock and halt the decoding task. We partition this task and perform it *on-the-fly* in order to speed up the total task latency and eliminate hardware components. We present results for applying this technique to an LDPC decoder for the IEEE 802.11n standard.**

***Keywords*—LDPC codes; iterative decoding; syndrome calculation; throughput enhancement**

I. Introduction

Modern communication standards are increasingly adopting low-density parity-check (LDPC) codes [1] as the choice for forward error correction. These codes display outstanding error-correction performance and are usually decoded by an iterative algorithm that allows the use of highly parallel architectures. High-throughput and low power operation are required for state-of-the-art mobile devices that may incorporate the use of these codes. Wireless communication and magnetic storage are among the most prominent target applications currently intended for such codes.

In this paper, we propose to optimize one recurrent task that is performed within each decoding iteration. Syndrome check or verification is performed in order to check the validity of the obtained codeblock and hence decide whether to continue or halt the decoding process. This task corresponds to the evaluation of all the parity-check constraints imposed by the parity-check matrix. We propose to perform this check *on-the-fly* so that a partially unsatisfied parity-check constraint can disable a potential useless syndrome verification on the entire parity-check matrix. We identify as benefits from this technique the elimination of several hardware elements, a reduction on the overall task latency and an increase on system throughput.

One form of the proposed technique has been identified in [2] for the purpose of improving the energy-efficiency of a decoder. This technique nevertheless is sub-optimal in the error-correcting sense as it introduces undetected codeblock errors. The contributions of our work include the assumptions for this technique and a performance analysis, along with a proposal to recover the performance loss. Furthermore, we identify the main benefit of such technique to be the enhancement of the system throughput.

The paper is organized as follows: Section II presents LDPC codes and the iterative decoding. Section III outlines the proposed syndrome check method and its performance. In Section IV the system level impact is presented along with results for a VLSI architecture. Section V concludes the paper.

II. LDPC Codes

Low-density parity-check codes are linear codes characterized by a sparse parity-check matrix $\boldsymbol{H}$. The number of nonzero elements in $\boldsymbol{H}$ is relatively small compared to the dimensions $M \times N$ of $\boldsymbol{H}$. For such matrix there are M parity-check constraints over N code symbols. A valid codeblock $\boldsymbol{c}$ satisfies the condition:

$$\boldsymbol{H} \cdot \boldsymbol{c}^T = \boldsymbol{S} = \boldsymbol{0} \,, \tag{1}$$

where $\boldsymbol{S}$ is referred to as the *syndrome*. Indeed the condition $\boldsymbol{S} = \boldsymbol{0}$ suggests that no further decoding iterations are necessary. Typically a maximum number of iterations is set to define an unsuccessful decoding operation. Several works like [3] and the references therein have proposed methods for early stopping of the decoding operation. These methods mainly monitor variables of the decoding process in order to make predictions about the possibility for convergence but introduce computational overheads and some loss in the error-correction performance.

These codes are usually decoded by iterative message-passing algorithms. In [4] a generalization for the decoding of sparse parity-check matrix codes was performed. This work consolidated several concepts that have greatly optimized the decoding process, such as a merger of messages to save on memory requirements and layered-scheduling that exploits so-called *architecture-aware* codes. The decoding algorithm is called the turbo-decoding message-passing (TDMP) algorithm. At the core of TDMP decoding lies a soft-input soft-output (SISO) message computation kernel. Several choices of SISO kernels offer tradeoffs on computational complexity and error-correction performance. In [5] a comparison is performed in terms of energy efficiency among the most prominent SISO kernels.

The messages involved in these SISO kernels are given in the form of log-likelihood ratios (LLRs). For every received code symbol (we address binary codes) x the corresponding LLR is given by:

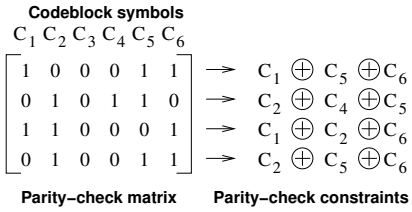


Fig. 1. Example parity-check constraints

$$L(x) = \log \frac{P(x=0)}{P(x=1)} \tag{2}$$

where $P(A = y)$ defines the probability that A takes the value y. An LLR in this context provides a measure of reliability (the LLR magnitude) of a decision on the value of x (the LLR sign). The decoding process starts with channel observations corresponding to each received code symbol in the form of LLRs, referred to as *intrinsic* messages. Throughout the decoding process new messages are calculated from independent code constraints, called *extrinsic* messages. During each decoding iteration the total sum of intrinsic and extrinsic messages provides the *posterior* messages, where a hard-decision upon each posterior message indicates the decoded code symbol.

III. Syndrome Check

A hard-decision vector on the posterior messages is required after each decoding iteration in order to calculate the syndrome. Syndrome calculation involves the product in equation (1), but this is equivalent to evaluate each parity-check constraint (each row in $\boldsymbol{H}$) with the corresponding code symbols. Figure 1 shows an example correspondance between the code symbols, the parity-check matrix and the parity-check constraints.

The parity-check constraints are usually of even parity and the $\oplus$ operation corresponds to the modulo-2 addition. The arguments of each constraint correspond to the hard-decision of each LLR (sign). A nonzero syndrome would correspond to any parity-check constraint resulting in odd parity. This condition suggests that a new decoding iteration must be triggered. The calculation of the syndrome in this way is synonymous to the verification of all parity-check constraints and indeed we refer to this as only *syndrome check*.

The typical syndrome check requires a separate memory for the hard-decision symbols, a separate unit for the syndrome calculation (or verification of parity-check constraints) and indeed consumes time in which no decoding is involved.

A. Proposed Method

Works like [6] and [7] have shown how the LLR values evolve within the decoding process. Depending upon the operating signal-to-noise ratio (SNR) regime these values will initially fluctuate or enter right away a strictly monotonic behavior. Figure 2 shows the simulated LLRs magnitude evolution of an instance of decoding the quasi-cyclic LDPC code defined in [8], for code length 648 and coding rate 1/2 over the AWGN channel at an SNR $E_b/N_0 = 1.5dB$.

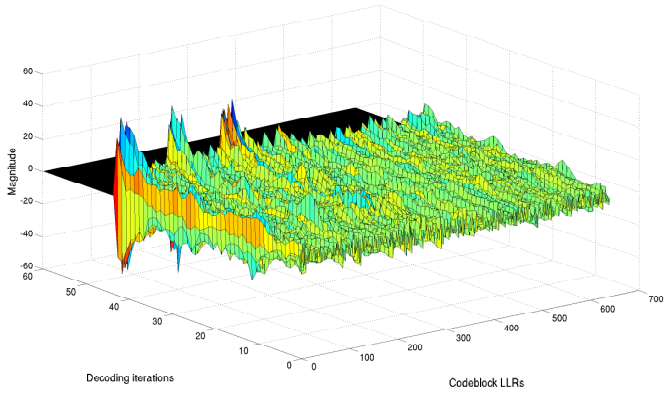


Fig. 2. LLRs magnitude evolution as a function of decoding iterations

Based upon the behavior of the LLRs we propose to perform the syndrome check *on-the-fly* in the following way: each parity-check constraint is verified right after each row is processed. Algorithm 1 outlines the proposed syndrome check within one decoding iteration for a parity-check matrix with M rows.

Algorithm 1 On-the-fly syndrome check

1. Decode each row i (or a plurality thereof for parallel architectures)
2. Evaluate each parity-check constraint PC_i by performing the $\oplus$ operation on the hard-decision values
3. Verification:

if $(PC_i = 1)$ **then**
  Disable further parity-checks verification
else
  if $(i = M)$ **then**
    Halt decoding: valid codeblock found
  end if
end if

Because of the structure of architecture-aware LDPC codes [9] and quasi-cyclic LDPC codes like the ones defined in [8] it is possible to process several rows of $\boldsymbol{H}$ in parallel. In this work we focus on the task that is performed between decoding iterations. For the proposed syndrome check there are two extreme cases regarding the latency between iterations. The worst-case scenario corresponds to the case when all individual parity-checks are satisfied but at least one from the last batch to process fails, in which case a new decoding iteration is triggered. The best-case scenario is when at least one of the first rows' parity-check fails, this disables further rows' parity-check verification and the next decoding iteration starts right after the end of the current one. The difference with the typical syndrome check is that it is always performed and it necessarily consumes more time as it involves the check of the entire $\boldsymbol{H}$. Figure 3 shows the timing visualization of these scenarios and the evident source for latency reduction of the decoding task.

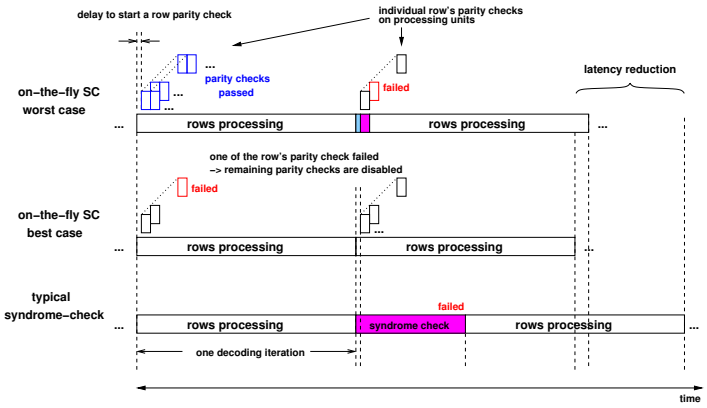


Fig. 3. Timing visualization for two consecutive decoding iterations

TABLE I
DECISION OUTCOMES OF THE PROPOSED SYNDROME CHECK

on-the-fly syndrome check	typical syndrome check	Outcome decision
Pass	Pass	Hit
Pass	Fail	False Alarm
Fail	Pass	Miss
Fail	Fail	Hit

B. Performance Analysis

A closer examination of the proposed syndrome check reveals the possibility for special scenarios. Indeed the proposed syndrome check does not correspond to equation (1) since the parity-check constraints are evaluated sequentially and their arguments (LLR sign) could change during the processing of the rows. Due to this there is a possibility where the decision taken by the *on-the-fly* strategy might not be the correct one at the end of the decoding process. Table I shows the possible outcomes of the decision taken by the proposed strategy in contrast to the typical syndrome check. A *Pass* condition is synonymous to the condition $\boldsymbol{S} = \mathbf{0}$. A *false alarm* outcome corresponds to the case when all parity-check constraints were satisfied, indeed halting the decoding task during any iteration as a valid codeblock has been identified (when in fact a final typical syndrome check would fail). On the other hand a *miss* outcome arises when during the last iteration (maximum iteration limit) a single parity-check constraint failed, rendering the codeblock as invalid (when in fact the typical syndrome check would pass). Both outcomes are the result of at least one LLR fluctuation right before the last row processing.

From this set of possible outcomes the probability P_H for the proposed syndrome check to be correct can be expressed by:

$$\begin{aligned} P_H &= 1-(P_{FA}+P_M) \\ &= 1-(P_P P_{CBE}+(1-P_P)(1-P_{CBE})) \end{aligned} \quad (3)$$

where P_{FA} is the probability of a *false alarm*, P_M is the probability of a *miss* , P_{CBE} is the probability of a codeblock error and P_P is the probability of the proposed syndrome check to pass.

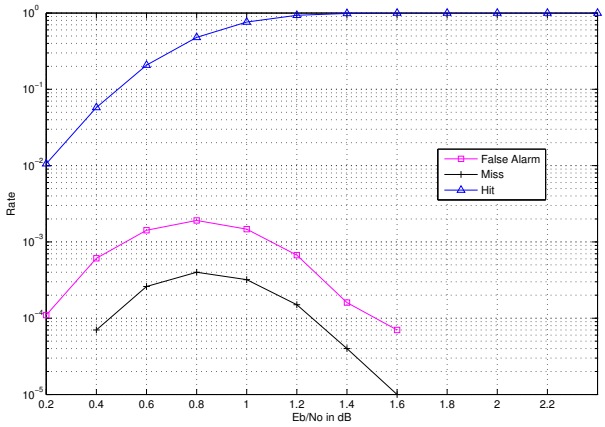


Fig. 4. Decision outcome rates from the proposed syndrome check

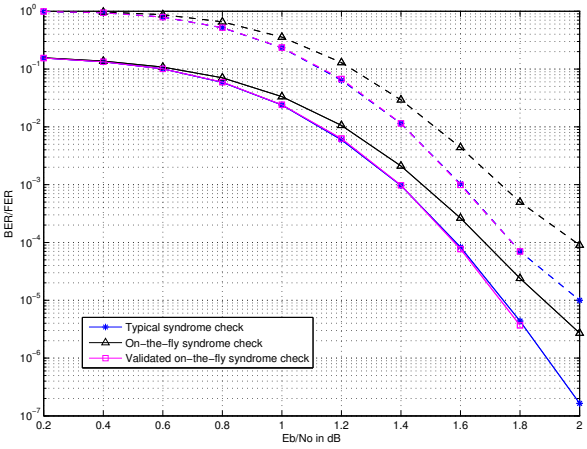


Fig. 5. Error-correction performance comparison

Based upon the analysis and observations by [6] and [7] the LLRs monotonic behavior is guaranteed for the high SNR regime, in this regime the outcome decision would be a *hit* with probability 1. Nevertheless as the SNR regime degrades the inherent fluctuations of the LLRs at the beginning of the decoding process may cause the decision to be a *miss* or a *false alarm* with nonzero probability. In Figure 4 we show the outcome of the decoding of 10^5 codeblocks using the same simulation scenario as in Figure 2 with code length 1944 and code rate 1/2 in order to observe the rate at which a *miss* and a *false alarm* may occur on the low SNR regime.

Even though the *hit* rate is at all times at least two orders of magnitude greater than a *miss* or a *false alarm* it is important to address their occurrence. A *miss* result would trigger an unnecessary retransmission in the presence of an automatic repeat request (ARQ) protocol, while a *false alarm* result would introduce undetected codeblock errors. This indeed represents some concerns that must be analyzed on an application-specific context, as for example a wireless modem for [8] is not likely to operate at such low SNR because of the required minimum packet-error rate performance.

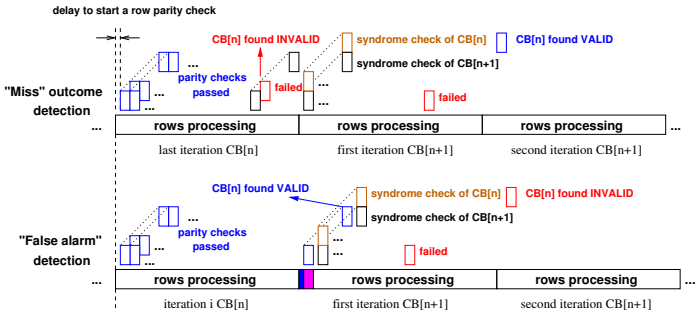


Fig. 6. *False alarm* and *miss* outcomes detection

The error-correction performance is affected by the *false alarm* outcomes. In Figure 5 we compare the simulated bit and frame error rates of the typical syndrome check and the proposed method, this corresponds to the same simulation scenario from Figure 4. Depending upon the target application the performance loss may not be tolerated, therefore we address the ways in which this situation can be circumvented.

Detection of the *miss* and *false alarm* outcomes can be performed in two ways:

1) Validating the result provided by *on-the-fly* syndrome check by calculating the typical syndrome check.
2) Allowing an outer coding scheme to detect such conditions: e.g., a cyclic redundancy check (CRC) that typically follows a codeblock decoding.

We propose to detect both *miss* and *false alarm* outcomes by validating the final calculated syndrome (in *on-the-fly* fashion) while executing the first iteration of the following codeblock. Figure 6 depicts both situations. In this way an ARQ protocol can react to a *false alarm* outcome and also avoid an unnecessary retransmission under the presence of a *miss* outcome. The performance is fully recovered, shown in Figure 5 as *validated on-the-fly* syndrome check.

IV. System Level Impact and Results

In Figure 7 we show a top level view for a canonical LDPC decoder based upon the decoding strategy proposed in [4]. From this it is evident that the proposed syndrome check strategy does not require dedicated elements for the syndrome calculation/verification step. In fact in order to implement the syndrome check in *on-the-fly* fashion each processing unit is augmented by a marginal set of components. Figure 8 shows a serial processing unit from the canonical decoder architecture driven by a SISO kernel, the added syndrome check capability is also shown. Synthesis results on CMOS $65nm$ technology showed that the area overhead due to the syndrome check capability is only 0.65% for a BCJR-based processing unit (the SISO kernel is the modified BCJR algorithm described in [4]).

If the *false alarm* and *miss* outcomes are to be detected by the proposed method in section III-B, then the syndrome check circuitry must be replicated and the hard-decision memory in Figure 7 must be kept. As can be observed the implementation of *on-the-fly* syndrome check must be evaluated

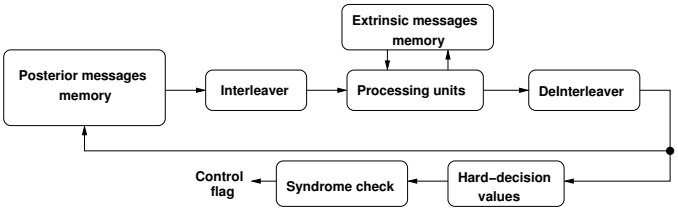


Fig. 7. Canonical decoder architecture

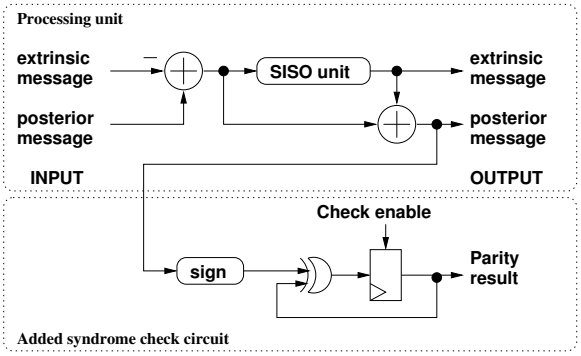


Fig. 8. Processing unit with syndrome check option

on an application-specific context: decoder operating range, outer multilevel coding schemes and ARQ protocols, logic and memory overheads, etc...

The main benefit of the proposed syndrome check is the speedup of the overall decoding task. The processing latency per decoding iteration for P processing units is given in number of cycles by:

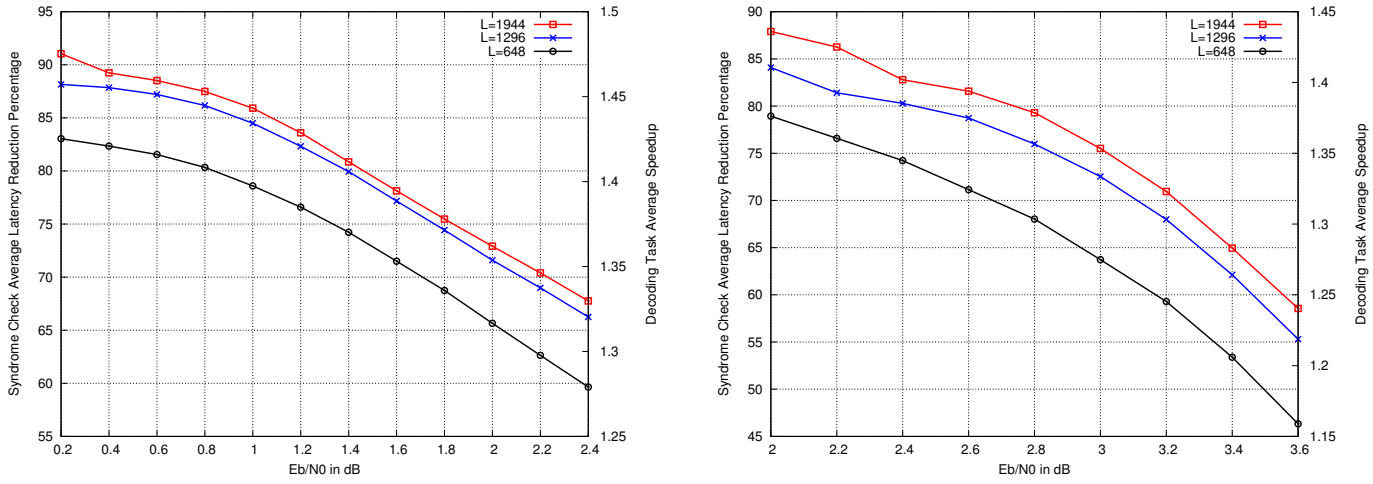
$$\tau_c = m_b \times \frac{Z}{P} \times L_c \tag{4}$$

where a quasi-cyclic LDPC code (e.g., in [8]) defines $\boldsymbol{H}$ as an array of m_b blockrows of Z rows per blockrow. In this case P rows are processed concurrently. L_c is the number of cycles consumed during the decoding task, where decoding and syndrome verification take place. This value depends upon the number of arguments to process per row, memory access latencies and syndrome verification duration. It is the latter time duration where our proposal exhibits advantages in terms of speedup. A reduction in the overall task latency improves as well the decoder throughput, given by:

$$\Gamma = \frac{N \times R \times f_{clk}}{I \times \tau_c} \tag{5}$$

where I is the total number of iterations, R the coding rate of the N code symbols and f_{clk} the operating frequency.

The main benefit from the proposed strategy is the reduction in the time consumed during the syndrome check when the decoding process is far from reaching convergence. It could be argued that the syndrome check may very well be disabled during a preset number of initial iterations, but still this tuning must be done offline or shall depend upon extraneous variables as the SNR. Estimating these variables provides sensible overheads. Figure 9 shows the obtained average latency reduction



(a) Rate 1/2

(b) Rate 5/6

Fig. 9. Average latency reduction for the syndrome check process and overall decoding task speedup

of the syndrome check process compared to the typical one as a function of operating SNR. A total of three use cases with different code lengths are shown, for a code rate of 1/2 in Figure 9a and code rate 5/6 in Figure 9b. The low SNR region provides the best opportunities for syndrome check latency reduction since LLRs fluctuate quite often in this region, i.e., a higher decoding effort renders useless the initial syndrome verification.

Indeed what this strategy is doing is speeding up a portion of the decoding task. With the use of Amdahl's law [10] it is possible to observe the overall speedup of the decoding task based upon the obtained latency reduction of the syndrome check. The overall speedup is a function of the fraction $P_{enhanced}$ of the task that is enhanced and the speedup $S_{enhanced}$ of such fraction of the task:

$$S_{overall} = \frac{1}{(1 - P_{enhanced}) + \frac{P_{enhanced}}{S_{enhanced}}} \quad (6)$$

Figure 9 shows as well the average speedup obtained as a function of operating SNR for the same test cases, these results consider that the syndrome check process corresponds to 35% of the overall decoding task per iteration. Amdahl's law provides an upper bound for the achievable overall speedup, 1.53 for this setup. The average speedup is higher for the code rate 1/2 case since the parity-check matrix contains more rows than the code rate 5/6. For the former case the achieved speedup ranged from 84% to 96% of the maximum achievable bound, this corresponds to enhancing the decoder throughput by a factor of 1.28 and 1.48 respectively.

V. Conclusion

An alternative method for performing the syndrome check of the iterative decoding of LDPC codes has been presented. By partitioning the calculation among the rows of the parity-check matrix several advantages have been identified. *On-the-fly* syndrome check reduces the number of hardware components on a VLSI architecture, offers a speedup in the overall decoding task and improves accordingly the decoding throughput. We analyzed the possible scenarios in which this technique may potentially provide erroneous outcomes regarding the validity of a codeblock and proposed how to handle these cases such that there is no error-correction performance loss. Such situations arise nevertheless at a very low rate and on the low SNR regime. Results from a decoder for the codes defined in IEEE 802.11n provided a speedup of up to a factor of 1.48 at a cost of less than 1% in logic area overhead for a $65nm$ CMOS process.

References

[1] R. Gallager, "Low-Density Parity-Check Codes," *IRE Trans. Inf. Theory*, vol. 7, pp. 21–28, January 1962.

[2] C. Struder, N. Preyss, C. Roth, and A. Burg, " Configurable High-Throughput Decoder Architecture for Quasi-Cyclic LDPC Codes ," in *Proceedings of the 42th Asilomar Conference on Signals, Systems and Computers* , October 2008.

[3] D. Shin, K. Heo, S. Oh, and J. Ha, " A Stopping Criterion for Low-Density Parity-Check Codes ," in *Proc. of IEEE VTC 2007-Spring* , 2007, pp. 1529–1533.

[4] M. Mansour, "A Turbo-Decoding Message-Passing Algorithm for Sparse Parity-Check Matrix Codes," *IEEE Trans. on Signal Processing*, vol. 54, no. 11, pp. 4376–4392, November 2006.

[5] E. Amador, V. Rezard, and R. Pacalet, " Energy Efficiency of SISO Algorithms for Turbo-Decoding Message-Passing LDPC Decoders ," in *Proc. of 17th IFIP/IEEE International Conference on Very Large Scale Integration* , October 2009.

[6] A. Heim and U. Sorger, " Turbo Decoding: Why Stopping-Criteria Do Work ," in *5th International Symposium on Turbo Codes and Related Topics* , 2008, pp. 255–259.

[7] G. Lechner and J. Sayir, " On the Convergence of Log-Likelihood Values in Iterative Decoding ," in *Mini-Workshop on Topics in Information Theory, Essen, Germany* , September 2002.

[8] IEEE-802.11n, "Wireless LAN Medium Access Control and Physical Layer Specifications: Enhancements for Higher Throughput," *P802.11n-2009*, October 2009.

[9] M. Mansour and N. Shanbhag, "High-Throughput LDPC Decoders," *IEEE Trans. on VLSI Systems*, vol. 11, no. 6, pp. 976–996, December 2003.

[10] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, third edition, 2003.