# Energy efficiency of SISO algorithms for turbo-decoding message-passing LDPC decoders

Erick Amador, Vincent Rezard, Renaud Pacalet

# Energy Efficiency of SISO Algorithms for Turbo-Decoding Message-Passing LDPC Decoders

Erick Amador
EURECOM
06904 Sophia Antipolis, France
erick.amador@eurecom.fr

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

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

***Abstract*—The decoding of LDPC codes using the turbo-decoding message-passing strategy is considered. This strategy can be used with different SISO message computation kernels. We analyze the suitability for VLSI implementation of various message computation algorithms in terms of implementation area, energy consumption and error-correcting performance. As one of the computation kernels, we introduce the recent Self-Corrected Min-Sum algorithm and show the advantages it brings from an energy efficiency perspective. We present comparisons among the studied kernels implemented in a $65nm$ CMOS process and use a test case from the codes defined in IEEE 802.11n to show differences in energy efficiency.**

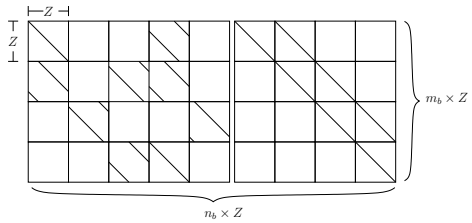
## I. Introduction

Low-density parity-check (LDPC) codes [1] currently stand as one of the best known error-correcting codes due to their capacity-approaching performance and the inherent parallelism of their iterative decoding algorithm. Several communication standards have already adopted these codes for forward-error correction. The implementation of these decoders present several challenges, especially when targeting wireless communications on mobile terminals, where ultra-low power operation is required.

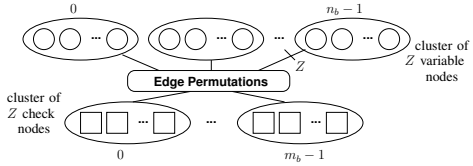
In this paper we investigate the energy efficiency of VLSI decoders based on the turbo-decoding message-passing (TDMP) [2] strategy using several message computation kernels. We perform a design-time exploration in order to assess the trade-offs between energy consumption, error-correcting performance and implementation area. We introduce the recent Self-Corrected Min-Sum algorithm [3] as a message computation kernel and demonstrate its native energy efficient capabilities. The concept of *erasing* unreliable messages in this algorithm reduces the net switching activity of the processing units, it allows a reduction of the number of active processing nodes per iteration and enables a simple stopping criterion for the iterative processing. We show results for a TDMP LDPC decoder for the codes defined in IEEE 802.11n [4] using a $65nm$ CMOS process. The paper is organized as follows: Section II introduces LDPC codes and TDMP decoding, Section III shows the message computation kernels explored and Section IV presents the decoder architecture proposed. Section V elaborates on the main points concerning the energy efficiency of these iterative decoders along with a comparison among the explored computation kernels. Section VI concludes the paper.

## II. LDPC Codes

LDPC codes are linear block codes defined by a sparse parity-check matrix $H$. The number of nonzero elements in a row and in a column of $H$ define the *degree* of the row and column respectively. The code can be represented also by a bipartite graph, where columns of $H$ are mapped to *variable* nodes and rows are mapped to *check* nodes. Nonzero elements in $H$ represent edges in the graph between the corresponding nodes. In order to simplify implementation issues a specific structure can be enforced within the code. Figure 1 shows the parity-check matrix and graph for a structured LDPC code.




(a) Parity-check matrix




(b) Bipartite graph

Fig. 1. Structured LDPC code

Structured codes consist of several layers of non-overlapping rows, where these layers are composed of $Z \times Z$ submatrices. $Z$ describes the level of parallelism within the code since $Z$ non-overlapping rows can be processed at the same time. These submatrices can be either a permutation matrix derived from an identity matrix or an all-zeros matrix. For the matrix shown in figure 1(a) there are $M = m_b \times Z$ parity-check constraints ($m_b$ block-rows) and $N = n_b \times Z$ codeword symbols ($n_b$ block-columns). The graph representation in figure 1(b) contains edges of width $Z$, grouping $Z$ nodes into clusters. This allows the possibility to instantiate a subset of the processing nodes, generating a clear advantage in terms of flexibility and silicon area as well as reducing the complexity of the interconnection network.

### A. TDMP Decoding

Gallager introduced an iterative two-phase message-passing decoding algorithm [1] where check nodes and variable nodes exchange extrinsic reliability values associated with each codeword symbol. Each decoding iteration consists of two phases: variable nodes update and send messages to the neighboring check nodes, and check nodes update and send back their corresponding messages. The operation of each node is independent and in general can be executed in parallel with other nodes. This characteristic enables the use of different scheduling techniques that impact the convergence speed of the decoding task. The TDMP schedule [2] shows important improvements over the two-phase schedule: a faster convergence and reduced memory requirements.

In TDMP the check nodes of the graph are evaluated sequentially, updating and propagating more reliable messages along the graph. Furthermore, this method merges check and variable updates in one step, reducing the memory requirements when compared to the traditional two-phase method. Another important advantage is the reduction in the number of iterations by up to 50%, indeed saving energy by the same proportion.

From the detailed description in [5] we summarize the TDMP decoding as follows. Let the vector $\boldsymbol{\delta} = [\delta_1, \ldots, \delta_N]$ denote the initial channel observations (*intrinsic* information) per codeword symbol as log likelihood ratios and a vector $\boldsymbol{\gamma} = [\gamma_1, \ldots, \gamma_N]$ denote the sum of all messages generated in the rows of $H$ for each codeword symbol (posterior messages). Let us define as well a vector $\boldsymbol{\lambda}^i = [\lambda_1^i, \ldots, \lambda_{c_i}^i]$ for each row $i$ in $H$ that contains the $c_i$ extrinsic messages generated after each decoding round, where $c_i$ is the degree of the row. Let $I_i$ define the set of $c_i$ nonzero values in row $i$, such that the $c_i$ elements of $\boldsymbol{\gamma}$ and $\boldsymbol{\delta}$ that participate in row $i$ are denoted by $\boldsymbol{\gamma}(I_i)$ and $\boldsymbol{\delta}(I_i)$ respectively. Furthermore, let the vector $\boldsymbol{\rho}$ define the prior messages. Algorithm 1 describes the decoding of the $i$th row of $H$.

**Algorithm 1** TDMP Decoding

Initialization
$\boldsymbol{\lambda}^i \leftarrow \mathbf{0}$
$\boldsymbol{\gamma}(I_i) \leftarrow \boldsymbol{\delta}(I_i)$
For each iteration:
1) Read vectors $\boldsymbol{\gamma}(I_i)$ and $\boldsymbol{\lambda}^i$
2) Generate prior messages: $\boldsymbol{\rho} = \boldsymbol{\gamma}(I_i) - \boldsymbol{\lambda}^i$
3) Process $\boldsymbol{\rho}$ with a soft-input soft-output (SISO) algorithm: $\boldsymbol{\Lambda} = SISO(\boldsymbol{\rho})$
4) Writeback vectors:
$\boldsymbol{\lambda}^i \leftarrow \boldsymbol{\Lambda}$
$\boldsymbol{\gamma}(I_i) \leftarrow \boldsymbol{\rho} + \boldsymbol{\Lambda}$

The process iterates until a stopping criterion is satisfied, refer to section V-C. Hard decisions are taken by slicing the vector $\boldsymbol{\gamma}$ to obtain the decoded message.

## III. SISO KERNELS

The processing complexity of the decoding task resides in the operations performed in the variable and check nodes of the code graph. Essentially the operation at the variable node is an addition of the incoming messages, whereas the operation at the check node involves more operations and it is where the tradeoff of performance and complexity takes place. In the context of TDMP the check node operation takes place on step 3 of algorithm 1. We refer to this step as *message computation* for generating the vector $\boldsymbol{\Lambda} = SISO(\boldsymbol{\rho})$ from the prior messages $\boldsymbol{\rho}$. The optimal message computation is performed by the Sum-Product (SP) algorithm [1] by:

$$\Lambda_j = \psi^{-1}\Big(\sum_{n \in I_i \setminus j} \psi(\rho_n)\Big) \tag{1}$$

where

$$\psi(x) = -\frac{1}{2}\log(\tanh(\frac{x}{2})) = \psi^{-1}(x) \tag{2}$$

Implementing (2) is highly complex mainly due to the effects of quantization and the nonlinearity of the function. Along with the TDMP schedule, [2] proposed the computation of messages by using a simplified form of the BCJR algorithm [6] to process the 2-state trellis of each single parity-check constraint of the code. Indeed this approach views an LDPC code as the parallel concatenation of single parity-check codes. The reader is refered to [2] and [5] for a detailed analysis. The computation of messages is performed by:

$$\Lambda_j = Q_{[j]}(\ldots(Q(Q(\rho_1, \rho_2), \rho_3), \ldots), \rho_{c_i}) \tag{3}$$

where

$$\begin{aligned} Q(x,y) \quad &= \max(x,y) + \max\left(\tfrac{5}{8} - \tfrac{|x-y|}{4}, 0\right) - \\ &\max\left(\tfrac{5}{8} - \tfrac{|x+y|}{4}, 0\right) - \max(x+y, 0) \end{aligned} \tag{4}$$

is the so-called *max-quartet* function and the subscript $[j]$ denotes the index of the variable to exclude from the computation.

The Min-Sum (MS) algorithm [7] approximates the operation in (1) with less complexity but at the cost of error-correcting performance. The MS operation computes messages by:

$$\Lambda_j = \Big(\prod_{n \in I_i \setminus j} (sign(\rho_n))\Big) \cdot \min_{n \in I_i \setminus j} |\rho_n| \tag{5}$$

Several correction methods have been proposed to recover the performance loss of the MS operation, such as the Normalized-MS (NMS) and Offset-MS (OMS) algorithms [7]. These correction methods essentially downscale the check node messages, which are overestimated in the first place in MS. NMS computes messages by scaling equation (5) by a factor $\alpha$, whereas OMS computes messages by:

$$\Lambda_j = \Big( \prod_{n \in I_i \setminus j} (sign(\rho_n)) \Big) \cdot \max \left( \min_{n \in I_i \setminus j} |\rho_n| - \beta, 0 \right) \quad (6)$$

where $\beta$ is an offset value. It has been shown recently in [3] that the sub-optimality of MS decoding is not due to the overestimation of the check node messages, but instead to the loss of the symmetric Gaussian distribution of these messages. This symmetry can be recovered by eliminating unreliable variable node messages or *cleaning* the inputs of the check node operation. By doing so [3] introduces the Self-Corrected MS (SCMS) decoding which exhibits quasi-optimal performance. An input to the check node operation is identified as *unreliable* if it has changed its sign with respect to the previous iteration. In algorithm 2 we show how to use SCMS to compute messages within TDMP for decoding row $i$.

**Algorithm 2** TDMP-SCMS

Initialization
$\boldsymbol{\lambda}^i \leftarrow \mathbf{0}$
$\boldsymbol{\gamma}(I_i) \leftarrow \boldsymbol{\delta}(I_i)$
At iteration $k \neq 0$:

1) Read vectors $\boldsymbol{\gamma}(I_i)$, $\boldsymbol{\lambda}^i$ and $\boldsymbol{\rho}^i_{old}$
2) Generate new prior messages: $\boldsymbol{\rho}^i_{new} = \boldsymbol{\gamma}(I_i) - \boldsymbol{\lambda}^i$
3) Generate MS input $\boldsymbol{\kappa}$ such that:

**for all** $j \in c_i$ **do**
  **if** $sign(\rho^i_{new_j}) \neq sign(\rho^i_{old_j})$ **then**
    $\kappa_j = 0$
  **else**
    $\kappa_j = \rho^i_{new_j}$
  **end if**
**end for**

4) Generate MS output $\boldsymbol{\Lambda} = MS(\boldsymbol{\kappa})$:

$$\Lambda_j = \Big( \prod_{n \in I_i \setminus j} (sign(\kappa_n)) \Big) \cdot \min_{n \in I_i \setminus j} |\kappa_n| \quad (7)$$

5) Writeback vectors:
$\boldsymbol{\lambda}^i \leftarrow \boldsymbol{\Lambda}$
$\boldsymbol{\rho}^i_{old} \leftarrow \boldsymbol{\rho}^i_{new}$
$\boldsymbol{\gamma}(I_i) \leftarrow \boldsymbol{\rho}^i_{new} + \boldsymbol{\Lambda}$

The vector $\boldsymbol{\kappa} = [\kappa_1, \ldots, \kappa_{c_i}]$ corresponds to the *corrected* inputs for the MS operation. Steps 3 and 4 correspond to the main features of the SCMS algorithm, where unreliable variable messages are identified and *erased*. In this way unreliable values are no longer propagated along the code graph.

Figure 2 shows the simulated error-correcting performance for these computation kernels applied to the TDMP decoding of the quasi-cyclic LDPC code (length $N = 1944$, rate $R = 1/2$) defined in IEEE 802.11n [4] over the AWGN channel with QPSK modulation. The normalization and offset values for NMS and OMS used were $\alpha = 0.8$ and $\beta = 0.35$ respectively, with a maximum number of 60 iterations. The

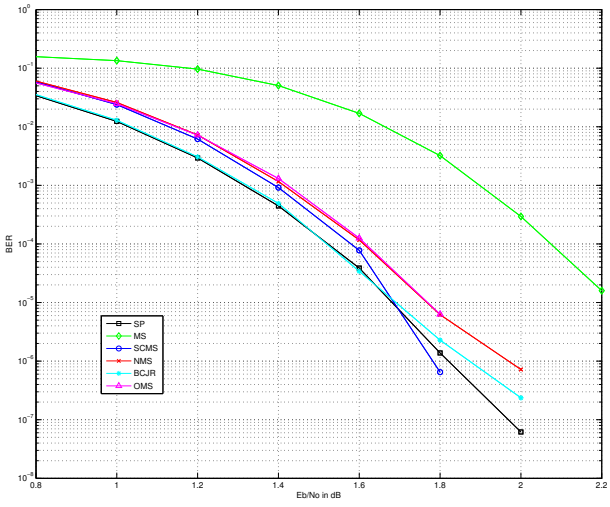



Fig. 2. Bit error rate performance

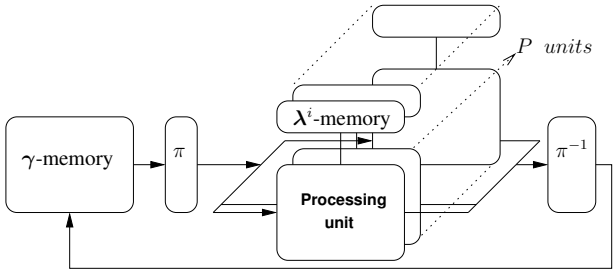



Fig. 3. TDMP decoder architecture

performance for BCJR is the closest to the optimal SP. MS performs the simplest approximation to SP and indeed shows the largest performance loss. Both NMS and OMS close significantly the performance gap between SP and MS but approach a relatively high error floor. SCMS is the MS-based kernel that shows the smallest performance gap to SP, in fact it shows a lower error floor, as reported in [3].

## IV. Decoder Architecture

Figure 3 shows a top level view of the proposed architecture for TDMP decoding. The required storage elements are the posterior messages memory ($\boldsymbol{\gamma}$) and the extrinsic messages memory ($\boldsymbol{\lambda}$). The $\boldsymbol{\gamma}$-memory stores as many values as the codeword length, whereas the $\boldsymbol{\lambda}$-memory size corresponds to the number of edges in the code graph. The $\boldsymbol{\lambda}$-memory is partitioned and bound to a processing unit following a static allocation of rows to processing units. Shuffling units $\pi$ and $\pi^{-1}$ are used to distribute posterior messages to and from the processing units. $P$ processing units handle the required operations for messages computation, figure 4 shows the data flow for this unit.

The SISO unit performs the message computation kernel of choice. As can be seen from section III the MS-based kernels fundamentally perform a running comparison of magnitudes and sign calculation. The processing unit for SCMS requires

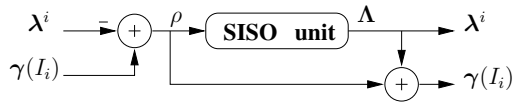



Fig. 4. SISO processing unit

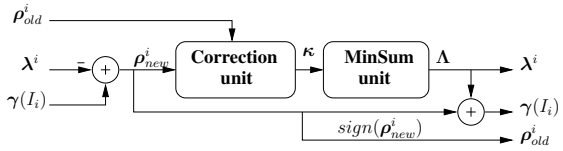



Fig. 5. SCMS processing unit

a precomputation block for performing step 3 from algorithm 2. Figure 5 shows the processing unit for SCMS message computation.

In table I we show the cell area from the synthesis of serial processing units using the different SISO kernels on a CMOS technology of $65nm$ and message quantization of 6-bits. We show as well the average energy per iteration consumed[1] when decoding the code $N = 1944$, $R = 1/2$ from [4]. It is interesting to notice the energy consumption of the SCMS unit compared to the other MS-based ones. It was observed that due to the *erased* messages used in SCMS this unit shows a net decrease in switching activity compared to the other MS-based units. As expected the BCJR unit shows the largest area and energy consumption, mainly due to the higher complexity of the datapath to implement equation (4).

TABLE I
COMPARISON OF PROCESSING UNITS

| Type | Cell area $[\mu m^2]$ | Energy $[nJ]$ N=1944,R=1/2 |
|---|---|---|
| MS | 3504.48 | 4.31 |
| NMS | 3827.04 | 6.71 |
| OMS | 3806.88 | 6.30 |
| SCMS | 3744.48 | 3.25 |
| BCJR | 5674.08 | 12.07 |

The architecture in figure 3 is independent of the SISO kernel used. Besides the processing units, only the $\lambda$-memory is affected by this choice. This memory stores as many values as edges in the code graph, but the MS-based algorithms allow a simple but valuable reduction in the memory size. These algorithms have the characteristic that for $n$ inputs there will be two output magnitudes and $n$ output signs. This allows to store only two magnitudes, $c_i$ signs and an index ($\log_2 c_i$ bits) for each row $i$. For our test case this corresponds to a reduction of 52% for this memory, this is indeed a main difference in implementation area between the BCJR and the MS-based kernels. This is not precisely the case with the SCMS kernel, as it requires to store the signs of the $c_i$ prior messages for each row $i$. This corresponds to an increase of 35% of this memory with respect to the other MS-based kernels.

[1]Pertinent activity files in VCD format along with the synthesized netlists were used with a power estimation tool.

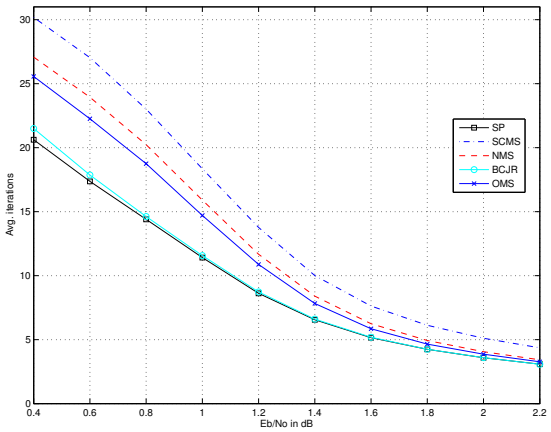



Fig. 6. Average decoding iterations

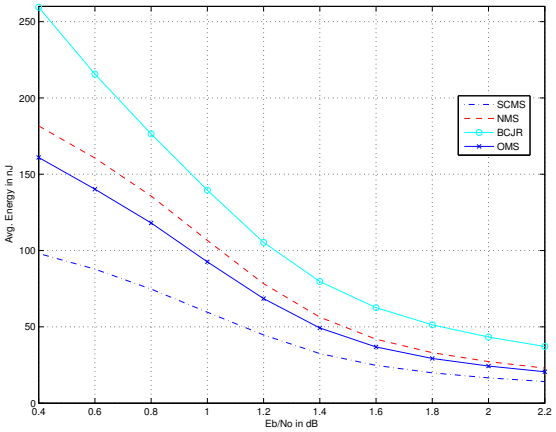



Fig. 7. Average energy on computation kernels

## V. Energy Efficiency

### A. Convergence Speed

The iterative nature of the decoding algorithm shows a dynamic behavior that depends upon external factors such as the received signal-to-noise ratio. In figure 6 we show the average number of iterations required for decoding using different computation kernels with the same simulation scenario from figure 2. The SP kernel is the fastest to converge, with BCJR following very closely the same behavior. OMS appears as the fastest MS-based kernel, followed by NMS and SCMS as the slowest one.

From an energy efficiency perspective it is compelling to compare the net consumption from each kernel, as for example the fastest one may consume more energy per iteration than the slowest one. In figure 7 we show the average energy consumed in the processing units accounting for the average number of iterations to complete the decoding task. It is observed how SCMS is the processing kernel with the best energy efficiency from the processing units perspective. BCJR is the kernel that requires the less number of iterations but each iteration incurs in a higher energy cost.

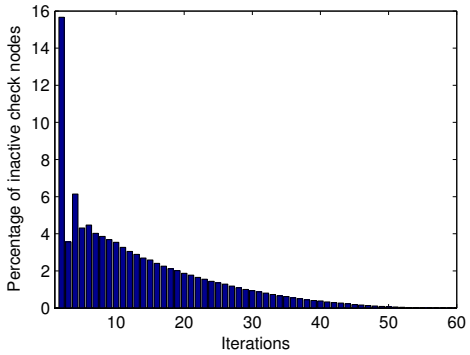



Fig. 8. Inactive check nodes in SCMS decoding

### B. Active nodes

The number of nodes in the code graph that are active during each decoding iteration impacts the energy consumption. In [8] the authors proposed to *deactivate* the variable nodes that have converged to a strong belief after a few iterations, where this condition is detected when the summation of all incoming messages surpasses a given threshold. The error-correcting performance is affected by the value of this threshold, furthermore this criterion adds a compare operation per variable node.

The SCMS kernel offers a simple criterion to disable a check node on a given iteration. The concept of *erasing* messages avoids the propagation of unreliable messages along the code graph. If there are two or more erased messages per row (step 3 in algorithm 2) that particular check node is effectively rendered useless for the decoding task as all its output values would have magnitude zero. Although there is a similar argument for any Min-Sum based kernel (two or more zero magnitude input messages) the SCMS kernel benefits from the fact that the minimum finders are not used.

Detecting this condition allows to save the processing required along with the writeback of messages to the memories. This criterion adds a compare operation per check node and does not affect the error-correcting performance. In figure 8 we show the average percentage of disabled check nodes per iteration for $10^5$ codewords at $E_b/N_0 = 1dB$ on our test case scenario. In our implementation this translates to an average reduction of 8% in energy per iteration for a SCMS-based decoder. Algorithm 3 outlines this simple criterion for energy reduction.

**Algorithm 3** Check node deactivation - SCMS

$\epsilon_i$: number of erased messages in row $i$
$M$: set of check nodes
**for all** rows $i \in M$ **do**
  Compute $\epsilon_i$
  **if** $\epsilon_i \geq 2$ **then**
    *Move to next row*
  **else**
    *Decode row*
  **end if**
**end for**

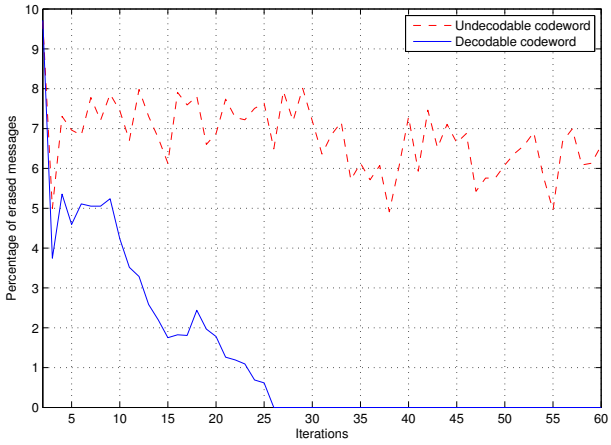



Fig. 9. Percentage of erased messages

### C. Stopping criterion

Iteration control oversees that only the necessary number of iterations are executed for both successful and unsuccessful decoding. Typically, successful decoding is verified by the condition $H \cdot x^T = 0$ (syndrome check), where $x$ is a valid codeword. If this condition is not satisfied decoding continues until a maximum number of iterations has been reached. Therefore early detection of an undecodable codeword is essential to save energy on unnecessary decoder operation.

We consider the case for early detection of an undecodable codeword and how SCMS provides this capability with the concept of *erased* messages at very low complexity. In SCMS the total number of erased messages per iteration approaches zero relatively fast for a decodable codeword. In the case of an undecodable codeword the number of erased messages fluctuates around a mean value. In figure 9 we show how the percentage of erased messages evolves with each iteration for a decodable and an undecodable codeword.

By detecting the characteristic decreasing monotonic behavior of the number of erased messages when the decoder enters a convergence state, it is possible to save energy on potential undecodable codewords. This simple criterion is equivalent in principle to the work in [9] but using a different decision metric: [9] follows the evolution of the mean checksum of check nodes, but requires knowledge of the SNR. The stopping criterion we propose for the SCMS kernel simply follows the evolution of the total number of erased messages by counting the increments of this metric and halting the decoding task once the number of increments exceeds a given threshold $T$. This threshold is a static parameter tuned by simulations and essentially trades error-correcting performance and the average number of iterations.

The implementation cost of this criterion corresponds to a counter along with an accumulator and a compare operation to make the halting decision.

In figure 10 we show the average number of total iterations for a few values of $T$ and the performance loss $L$ for a bit error

rate of $10^{-6}$. It can be observed that for $T = 8$ the average number of iterations can be reduced up to 60%, reducing energy consumption for the average decoding task by the same proportion.

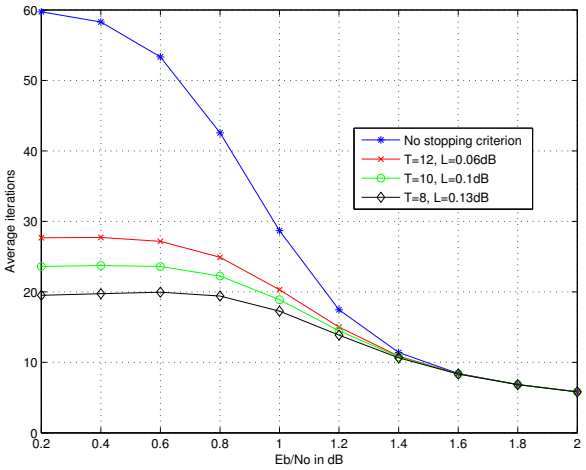



Fig. 10. Average total iterations and stopping criteria

### D. Decoders Comparison

We performed an estimation on the decoders implementation area and energy consumption of the individual components shown in the architecture of figure 3, using three computation kernels for the test case of the code $N = 1944, R = 1/2$ in [4]. Benes networks were used as shuffling units [10], and low-power dual-port RAMs for the required memories providing a bandwidth of 12 samples/cycle to three processing units in all decoders.

In figure 11 we show the implementation area and energy breakdown. The energy breakdown corresponds to the completion of a decoding task using the average number of iterations from $10^5$ codewords at $E_b/N_0 = 1dB$. To have a fair comparison all decoders used syndrome check as stopping criterion and all nodes activated at run-time. On all decoders at least 70% of the energy is consumed on the memory subsystem, this shows the relevance of the optimization that can be exploited by SCMS shown in section V-B.

Since the number of required iterations is highly dynamic, in table II we show the energy efficiency per iteration for these decoders in order to have a proper figure of merit.

TABLE II
ENERGY EFFICIENCY OF SISO KERNELS

| Energy efficiency $[pJ/bit/iteration]$ | BCJR | OMS | NMS | SCMS |
|---|---|---|---|---|
| | 64.87 | 46.98 | 47.61 | 43.61 |

## VI. CONCLUSIONS

We performed a comparison among several message computation kernels within TDMP decoding of LDPC codes to observe the tradeoffs on energy efficiency, implementation area and error-correcting performance. We proposed the use of the SCMS kernel and identified its built-in characteristics for better energy efficiency: reduced net switching activity in the processing units, reduced number of active nodes per iteration and a stopping criterion for early detection of undecodable codewords. Even though the differences on implementation area are small, OMS presented the smallest footprint and best energy efficiency for the average complete decoding task for the studied test case. Nevertheless SCMS shows a better error-correcting performance and energy efficiency per iteration, in addition to its built-in energy optimizations derived from the concept of erasing unreliable messages.

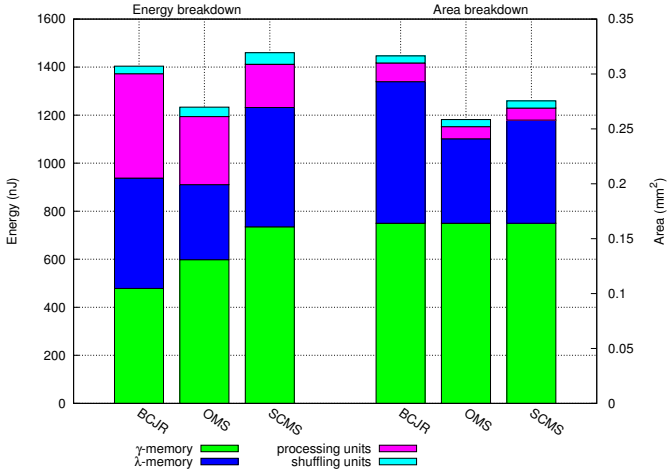



Fig. 11. Energy and area breakdown

## REFERENCES


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

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

[3] V. Savin, " Self-Corrected Min-Sum Decoding of LDPC Codes ," in *Proc. of IEEE International Symposium on Information Theory* , 2008, pp. 146–150.

[4] IEEE-802.11n, "Wireless LAN Medium Access Control and Physical Layer Specifications: Enhancements for Higher Throughput," *P802.11n/D3.07*, March 2008.

[5] Mohammad 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.

[6] L. Bahl, J. Cocke, F. Jelinek, and J. Raviv, "Optimal decoding of linear codes for minimizing symbol error rate (corresp.)," *Information Theory, IEEE Transactions on*, vol. 20, no. 2, pp. 284–287, 1974.

[7] J. Chen and M. Fossorier, " Near Optimum Universal Belief Propagation based Decoding of LDPC codes ," in *IEEE Trans. on Comm.* , 2002, pp. 406–414.

[8] E. Zimmermann, P. Pattisapu, K. Bora, and G. Fettweis, "Reduced Complexity LDPC Decoding using Forced Convergence," *Proc. of the 7th International Symposium on Wireless Personal Multimedia Communications*, pp. 12–15, September 2004.

[9] Zhiaqiang Cui, Lupin Chen, and Zhongfeng Wang, " An Efficient Early Stopping Scheme for LDPC Decoding ," in *13th NASA Symposium on VLSI Design* , 2007.

[10] M. Rovini, G. Gentile, and L. Fanucci, " Multi-size circular shifting networks for decoders of structured LDPC codes ," *IEE Electronics Letters* , pp. 938–940, August 2007.