Efficient Implementations of the Sum-Product Algorithm for Decoding LDPC Codes

Xiao–Yu Hu, Evangelos Eleftheriou, Dieter–Michael Arnold, and Ajay Dholakia
IBM Research, Zurich Research Laboratory, CH-8803 Rüschlikon, Switzerland

***Abstract*— Efficient implementations of the sum-product algorithm (SPA) for decoding low-density parity-check (LDPC) codes using log-likelihood ratios (LLR) as messages between symbol and parity-check nodes are presented. Various reduced-complexity derivatives of the LLR-SPA are proposed. Both serial and parallel implementations are investigated, leading to trellis and tree topologies, respectively. Furthermore, by exploiting the inherent robustness of LLRs, it is shown, via simulations, that coarse quantization tables are sufficient to implement complex *core operations* with negligible or no loss in performance. The unified treatment of decoding techniques for LDPC codes presented here provides flexibility in selecting the appropriate design point in high-speed applications from a performance, latency, and computational complexity perspective.**

I. Introduction

Iterative decoding of binary low-density parity-check (LDPC) codes using the sum-product algorithm (SPA) has recently been shown to approach the capacity of the additive white Gaussian noise (AWGN) channel within 0.0045 dB [1-3]. Efficient hardware implementation of the SPA has become a topic of increasing interest. The direct implementation of the original form of SPA has been shown to be sensitive to quantization effects [4]. In addition, using likelihood ratios can substantially reduce the required quantization levels [4]. A simplification of the SPA that reduces the complexity of the parity-check update at the cost of some loss in performance was proposed in [5]. This simplification has been derived by operating in the log-likelihood domain. Recently, a new reduced-complexity decoding algorithm that also operates entirely in the log-likelihood domain was presented [6]. It bridges the gap in performance between the optimal SPA and the simplified approach in [5]. Finally, low complexity software and hardware implementations of an iterative decoder for LDPC codes suitable for multiple access applications were presented in [7].

Here we present efficient implementations of the SPA and describe new reduced-complexity derivatives thereof. In our approach, log-likelihood ratios (LLR) are used as messages between symbol and parity-check nodes. It is known that in practical systems, using LLRs offers implementation advantages over using probabilities or likelihood ratios, because multiplications are replaced by additions and the normalization step is eliminated. The family of LDPC decoding algorithms presented here is called LLR-SPA.

The unified treatment of decoding techniques for LDPC codes presented here provides flexibility in selecting the appropriate design point in high-speed applications from a performance, latency, and computational complexity perspective. In particular, serial and parallel implementations are investigated, leading to trellis and tree topologies, respectively. In both cases, specific *core operations* similar to the special operations defined in the log-likelihood algebra of [8] are used. This formulation not only leads to reduced complexity LDPC decoding algorithms that can be implemented with simple comparators and adders but also provides the ability to compensate the loss in performance by using simple look-up tables or constant correction terms.

The remainder of the paper is organized as follows. In Section II, the SPA in the log-likelihood domain is described, and the issues associated with a brute-force implementation are discussed. In Section III, a trellis topology for carrying out the parity-check updates is derived. The core operation on this trellis is the LLR of the exclusive OR (XOR) function of two binary independent random variables [8], rather than the hyperbolic tangent operation used in the brute-force implementation. This core operation can either be implemented very accurately by using the $\max^*$ operation [9] or approximately by using the so-called sign-$\min$ operation. In either case, the check-node updates can be efficiently implemented on the trellis by the well-known forward–backward algorithm. Section IV is devoted to parallel processing, and a simple tree topology with a new core operation is proposed. It is shown that such an implementation offers smaller latency compared to the serial implementation. In practice, this core operation can be realized by employing a simple eight-segment piecewise linear function. In Section V, simulation results are presented, comparing the performance of the various alternative implementations of the LLR-SPA. Finally, Section VI contains a summary of the results and conclusions.

II. SPA in the log-likelihood domain

A binary (N, K) LDPC code [1, 2] is a linear block code described by a sparse $M \times N$ parity-check matrix H, i.e., H has a low density of 1s. The parity-check matrix H can be viewed as a bipartite graph with two kinds of nodes: N symbol nodes corresponding to the encoded symbols, and M parity-check nodes corresponding to the parity checks represented by the rows of the matrix H. The connectivity of the bipartite graph is such that the parity-check matrix H is its incidence matrix. For regular LDPC codes, each symbol is connected to d_s parity-check nodes and each parity-check node is connected to d_c symbol nodes. For irregular LDPC codes, d_s and/or d_c are not constant.

Following a notation similar to [2, 5], let $\mathcal{M}(n)$ denote the set of check nodes connected to symbol node n, i.e., the positions of 1s in the n-th column of the parity-check matrix H, and let $\mathcal{N}(m)$ denote the set of symbol nodes that participate in the m-th parity-check equation, i.e., the positions of 1s in the m-th row of H. Furthermore, $\mathcal{N}(m)\backslash n$ represents the set $\mathcal{N}(m)$, excluding the n-th symbol node, and similarly,

0-7803-7206-9/01/$17.00 © 2001 IEEE

$\mathcal{M}(n)\backslash m$ represents the set $\mathcal{M}(n)$, excluding the m-th check node.

In addition, $q_{n\to m}(x)$, $x \in \{0,1\}$, denotes the message that symbol node n sends to check node m indicating the probability of symbol n being 0 or 1, based on all the checks involving n except m. Similarly, $r_{m\to n}(x)$, $x \in \{0,1\}$, denotes the message that the m-th check node sends to the n-th symbol node indicating the probability of symbol n being 0 or 1, based on all the symbols checked by m except n. Finally, $\mathbf{y} = [y_1, y_2, \ldots, y_N]$ denotes the received word corresponding to the transmitted codeword $\mathbf{u} = [u_1, u_2, \ldots, u_N]$.

The LLR of a binary valued random variable U is defined as

$$L(U) \stackrel{def}{=} \log \frac{P(U=0)}{P(U=1)},$$

where $P(U=x)$ denotes the probability that the random variable U takes the value x. Furthermore, let us define the LLRs $\lambda_{n\to m}(u_n) \stackrel{def}{=} \log(q_{n\to m}(0)/q_{n\to m}(1))$ and $\Lambda_{m\to n}(u_n) \stackrel{def}{=} \log(r_{m\to n}(0)/r_{m\to n}(1))$. The LLR-SPA is then summarized as follows.

Initialization: Each symbol node n is assigned an *a posteriori* LLR $L(u_n) = \log\{P(u_n = 0|y_n)/P(u_n = 1|y_n\}$. In case of equiprobable inputs on an AWGN channel, $L(u_n) = 2y_n/\sigma^2$, where σ^2 is the noise variance. For every position (m,n) such that $H_{m,n} = 1$,

$$\begin{aligned} \lambda_{n\to m}(u_n) &= L(u_n), \\ \Lambda_{m\to n}(u_n) &= 0. \end{aligned}$$

Step (i) (check-node update): For each m, and for each $n \in \mathcal{N}(m)$, compute

$$\Lambda_{m\to n}(u_n) = 2\tanh^{-1}\left\{\prod_{n'\in\mathcal{N}(m)\backslash n} \tanh[\lambda_{n'\to m}(u_{n'})/2)]\right\}. \quad (1)$$

Step (ii) (symbol-node update): For each n, and for each $m \in \mathcal{M}(n)$, compute

$$\lambda_{n\to m}(u_n) = L(u_n) + \sum_{m'\in\mathcal{M}(n)\backslash m} \Lambda_{m'\to n}(u_n).$$

For each n, compute

$$\lambda_n(u_n) = L(u_n) + \sum_{m\in\mathcal{M}(n)} \Lambda_{m\to n}(u_n).$$

Step (iii) (decision): Quantize $\hat{\mathbf{u}} = [\hat{u}_1, \hat{u}_2, \ldots, \hat{u}_N]$ such that $\hat{u}_n = 0$ if $\lambda_n(u_n) \geq 0$, and $\hat{u}_n = 1$ if $\lambda_n(u_n) < 0$. If $\hat{\mathbf{u}}H^T = \mathbf{0}$, then halt the algorithm with $\hat{\mathbf{u}}$ as the decoder output; otherwise go to *Step (i)*. If the algorithm does not halt within some maximum number of iterations, then declare a decoder failure.

The check-node updates are computationally the most complex part of the LLR-SPA. Two issues influence their complexity: i) the topology used in computing the messages that a particular check node sends to the symbol nodes associated with it, and ii) the implementation of the core operation needed for computing these messages. For example, the core operation of the check-node update computation in *Step (i)* above is the hyperbolic tangent function, which is known to be difficult to implement in hardware. Furthermore, in a brute-force implementation of the check-node update (1), $d_c(d_c-1)$ multiplications are necessary per check node, with all multiplicands requiring the evaluation of the hyperbolic tangent core operation. Clearly, the higher the rate of the code, the higher the row degree d_c, thus leading to a higher number of multiplications. Therefore, the brute-force topology and its corresponding core operation are not suited for high-speed digital applications.

III. Serial Implementation: Trellis Topology

A. Check-Node Updates

Consider a particular check node m with d_c connections from symbol nodes in $\mathcal{N}(m) = (n_1, n_2, \ldots, n_{d_c})$. The incoming messages are then $\lambda_{n_1\to m}(u_{n_1})$, $\lambda_{n_2\to m}(u_{n_2}), \ldots, \lambda_{n_{d_c}\to m}(u_{n_{d_c}})$. The goal is to efficiently compute the outgoing messages $\Lambda_{m\to n_1}(u_{n_1})$, $\Lambda_{m\to n_2}(u_{n_2}), \ldots, \Lambda_{m\to n_{d_c}}(u_{n_{d_c}})$.

Let us define two sets of auxiliary binary random variables $f_1 = u_{n_1}, f_2 = f_1 \oplus u_{n_2}$, $f_3 = f_2 \oplus u_{n_3}, \ldots, f_{d_c} = f_{d_{c-1}} \oplus u_{n_{d_c}}$, and $b_{d_c} = u_{n_{d_c}}$, $b_{d_{c-1}} = b_{d_c} \oplus u_{n_{d_c-1}}, \ldots, b_1 = b_2 \oplus u_{n_1}$, where $\oplus$ denotes the binary XOR operation. It can easily be seen that for statistically independent binary random variable U and V [8],

$$L(U \oplus V) = \log \frac{1 + e^{L(U)+L(V)}}{e^{L(U)} + e^{L(V)}}. \quad (2)$$

Using (2) repeatedly, we can obtain $L(f_1), L(f_2), \ldots, L(f_{d_c})$ and $L(b_1), L(b_2), \ldots, L(b_{d_c})$ in a *recursive* manner based on the knowledge of $\lambda_{n_1\to m}(u_{n_1})$, $\lambda_{n_2\to m}(u_{n_2}), \ldots, \lambda_{n_{d_c}\to m}(u_{n_{d_c}})$. Using the parity-check node constraint $(u_{n_1} \oplus u_{n_2} \oplus \ldots \oplus u_{n_{d_c}}) = 0$, we obtain $u_{n_i} = (f_{i-1} \oplus b_{i+1})$ for every $i \in \{2, 3, \ldots, d_c - 1\}$. Therefore, the outgoing message from the check node m can be simply expressed as

$$\begin{aligned} \Lambda_{m\to n_i}(u_{n_i}) &= L(f_{i-1} \oplus b_{i+1}), \quad i = 2, 3, \ldots, d_c - 1, \\ \Lambda_{m\to n_1}(u_{n_1}) &= L(b_2), \\ \Lambda_{m\to n_{d_c}}(u_{n_{d_c}}) &= L(f_{d_c-1}). \end{aligned} \quad (3)$$

The total computational load consists of the forward recursive computation of $L(f_i)$, the backward recursive computation of $L(b_i)$, and the final pairwise part in (3), which amounts to $3(d_c-2)$ core operation of the type $L(U \oplus V)$ per check node. This should be compared to $d_c(d_c-1)$ hyperbolic tangent operations for the check-node updates of the brute-force topology. Clearly, the above procedure is exactly the forward–backward algorithm on a single-state trellis, as shown in Fig. 1. The serial nature of computations makes the latency in computing a check-node update of the order $O(d_c)$.

B. Symbol-Node Updates

In the log-likelihood domain, the symbol-node updates consist only of additions of incoming messages. It is more conve-

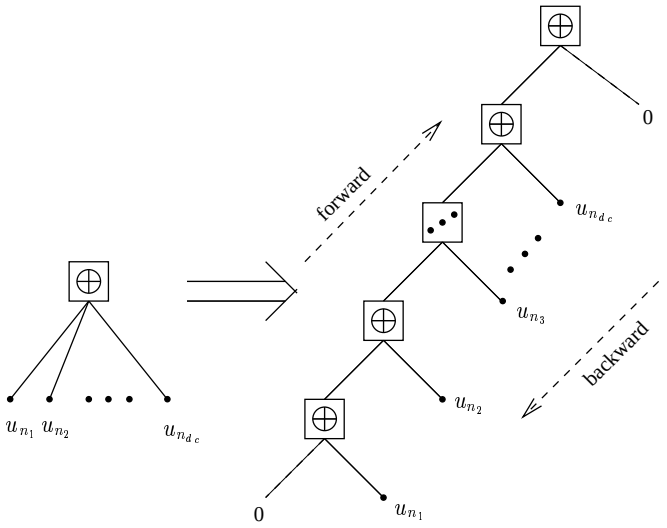

Fig. 1. Serial configuration for computing check-node updates.

nient to compute the posterior LLR for the symbol u_n, given by

$$\lambda_n(u_n) = L(u_n) + \sum_{i=m_1}^{m_{d_s}} \Lambda_{m_i \to n}(u_n),$$

where $\Lambda_{m_i \to n}(u_n)$, $i = m_1, \ldots, m_{d_s}$ are the incoming LLRs from the parity-check nodes $\mathcal{M}(n) = (m_1, m_2, \ldots, m_{d_s})$ connected to the symbol node n. Then, the outgoing messages from symbol node n are obtained as

$$\lambda_{n \to m_i}(u_n) = \lambda_n(u_n) - \Lambda_{m_i \to n}(u_n), \quad i = m_1, \ldots, m_{d_s}. \tag{4}$$

The total computational load for a symbol-node update is $2d_s + 1$ additions. Note that this computational complexity figure includes the number of operations needed to obtain the posterior LLR use in *Step (iii)* of LLR-SPA.

C. Efficient Implementation of Core Operation $L(U \oplus V)$

In this section, two efficient implementation versions of the core operation $L(U \oplus V)$ are described, both of which are amenable to efficient VLSI design.

The first version is analogous to the $\max^*$ operation used in turbo codes [9, 10]. By using the Jacobian logarithm twice, we obtain

$$\begin{aligned} L(U \oplus V) &= \log \frac{1 + e^{L(U)+L(V)}}{e^{L(U)} + e^{L(V)}} \\ &= \log[1 + e^{L(U)+L(V)}] - \log[e^{L(U)} + e^{L(V)}] \\ &= \max[0, L(U) + L(V)] + \log(1 + e^{-|L(U)+L(V)|}) \\ &\quad - \max[L(U), L(V)] - \log(1 + e^{-|L(U)-L(V)|}). \end{aligned}$$

It can be shown that the following equality holds:

$$\begin{aligned} &\max[0, L(U) + L(V)] - \max[L(U), L(V)] \\ &= \text{sign}[L(U)]\text{sign}[L(V)] \cdot \min[|L(U)|, |L(V)|]. \end{aligned}$$

Therefore,

$$\begin{aligned} L(U \oplus V) &= \text{sign}[L(U)]\text{sign}[L(V)] \cdot \min[|L(U)|, |L(V)|] \\ &\quad + \log[1 + e^{L(U)+L(V)}] \\ &\quad - \log[e^{L(U)} + e^{L(V)}], \end{aligned} \tag{5}$$

in which the terms $\log(1 + e^{-|L(U)+L(V)|})$ and $\log(1 + e^{-|L(U)-L(V)|})$ can be implemented by a look-up table. Fig. 2 shows a plot of the function $g(x) = \log(1 + e^{-|x|})$. A 3-bit coarse quantization table of $g(x)$ is given in Table I. The maximum approximation error is less than 0.05.

The function $g(x)$ can also be approximated more accurately by a piece-wise linear function where the multiplying factors are powers of two and therefore simple to implement in hardware with shift operations. Table II shows a piece-wise linear approximation of $g(x)$ with only eight regions. Fig. 2 shows the corresponding piece-wise linear approximation plot. As can be seen, the piece-wise linear function offers almost a perfect match to the original function. In summary, the core operation $L(U \oplus V)$ can be realized using four additions, one

TABLE I

QUANTIZATION TABLE FOR $g(x) = \log(1 + e^{-|x|})$.

$\lvert x\rvert$	$\log(1 + e^{-\lvert x\rvert})$	$\lvert x\rvert$	$\log(1 + e^{-\lvert x\rvert})$
[0, 0.196)	0.65	[1.05, 1.508)	0.25
[0.196, 0.433)	0.55	[1.508, 2.252)	0.15
[0.433, 0.71)	0.45	[2.252, 4.5)	0.05
[0.71, 1.05)	0.35	[4.5, $+\infty$)	0.0

TABLE II

PIECEWISE LINEAR FUNCTION APPROXIMATION FOR $g(x) = \log(1 + e^{-|x|})$.

$\lvert x\rvert$	$\log(1 + e^{-\lvert x\rvert})$	$\lvert x\rvert$	$\log(1 + e^{-\lvert x\rvert})$
[0, 0.5)	$-\lvert x\rvert * 2^{-1} + 0.7$	[2.2, 3.2)	$-\lvert x\rvert * 2^{-4} + 0.2375$
[0.5, 1.6)	$-\lvert x\rvert * 2^{-2} + 0.575$	[3.2, 4.4)	$-\lvert x\rvert * 2^{-5} + 0.1375$
[1.6, 2.2)	$-\lvert x\rvert * 2^{-3} + 0.375$	[4.4, $+\infty$)	0.0

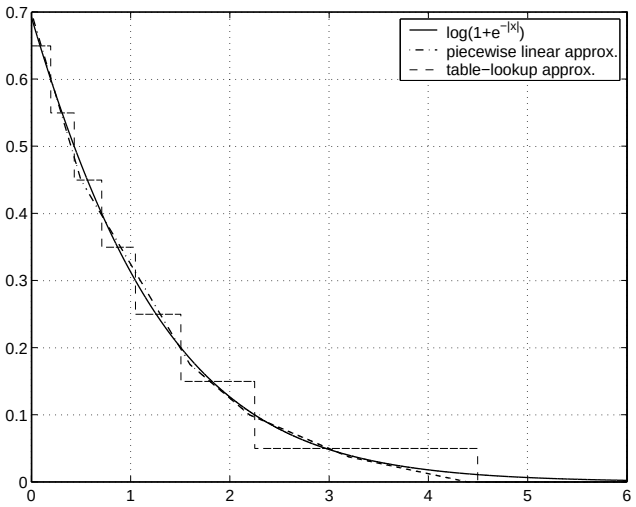

Fig. 2. The function $g(x) = \log(1 + e^{-|x|})$.

comparison, and two *corrections*. Each correction itself can be a table look-up operation or a linear function evaluation with a shift and a constant addition.

It can readily be seen that the core operation $L(U \oplus V)$ can also be approximated as [8]

$$\begin{aligned} L(U \oplus V) &= \log \frac{1+e^{L(U)+L(V)}}{e^{L(U)}+e^{L(V)}} \\ &\approx \text{sign}[L(U)]\text{sign}[L(V)] \\ &\quad \cdot \min[|L(U)|, |L(V)|], \end{aligned} \tag{6}$$

which is called herein the sign-min approximation. The advantage of using the sign-min approximation lies in its simplicity. No additions are needed for check-node updates, merely two-way comparisons, hence requiring a very small number of logic gates.

Finally, the difference between the exact $L(U \oplus V)$ operation and its sign-min approximation is given by the term $\log(1+e^{-|L(U)+L(V)|}) - \log(1+e^{-|L(U)-L(V)|})$, called the *correction factor* in [6]. This correction factor can be described by the bivariate function

$$s(x,y) = \log \frac{1+e^{-|x+y|}}{1+e^{-|x-y|}}, \tag{7}$$

where the arguments x and y represent the LLRs $L(U)$ and $L(V)$, respectively. It is shown in [6] that this correction factor can be approximated by a single constant without incurring any loss in performance with respect to the SPA. Clearly, one can also use the function $g(x)$ shown in Fig. 2 instead of the bivariate function (7), introduced in [6], to determine a correction factor. For example, let $x_1 = L(U) + L(V)$ and $x_2 = L(U) - L(V)$. Then, a simple rule similar to the one proposed in [6] is

$$g(x_1) - g(x_2) = \begin{cases} c & \text{if } |x_1| < 2 \text{ and } |x_2| > 2|x_1| \\ -c & \text{if } |x_2| < 2 \text{ and } |x_1| > 2|x_2| \\ 0 & \text{otherwise.} \end{cases} \tag{8}$$

This means that the correction factor is zero when the values $g(x_1)$ and $g(x_2)$ are close to each other. Otherwise, depending on the relative magnitude of the values $g(x_1)$ and $g(x_2)$, the correction factor is a positive or a negative nonzero value determined according to the signal-to-noise ratio. In this case, the computational complexity of the $L(U \oplus V)$ core operation is a single two-way comparison and an addition with a constant.

IV. Parallel Implementation: Tree Topology

For applications with high throughput requirements, recursive algorithms such as the forward–backward algorithm may not be well suited. In this section, a simple tree topology that enables fast check-node updates is described. The symbol-node updates remain the same as in (4).

We begin by defining an auxiliary binary random variable $S_m = \sum_{i=1}^{d_c} \oplus u_{n_i}$. The LLR of S_m at a particular check node m can be computed using the tree topology shown in Fig. 3.

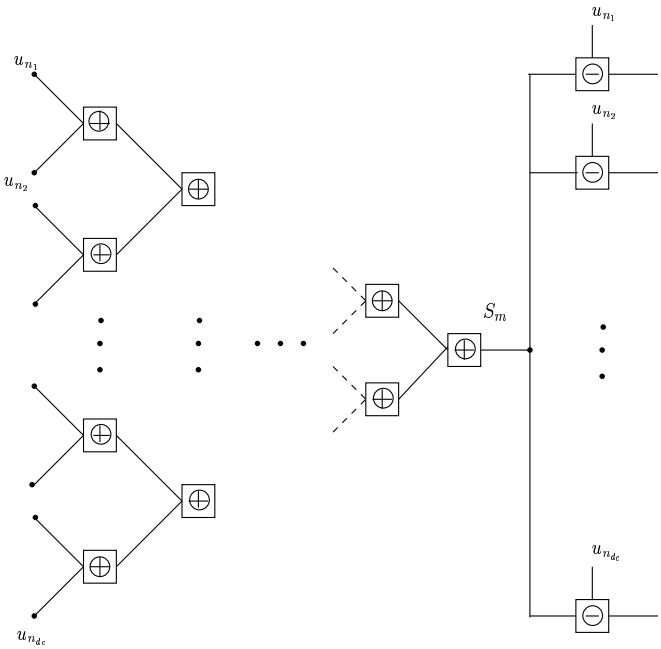

Fig. 3. Parallel configuration for computing check-node updates.

The operation at each node in the tree is $L(U \oplus V)$, which can be efficiently implementated using any of the alternatives described in Section III-C. The latency in computing the LLR of S_m is of order $O(\log d_c)$, resulting in a speed-up factor of $O[d_c/\log(d_c)]$ compared to the serial trellis topology of Section III-A.

Having obtained the LLR of S_m, we now describe a simple and efficient way to compute the outgoing LLRs $\Lambda_{m\to n_i}(u_{n_i})$. Let us consider

$$\begin{aligned} L(S_m) &= L(\sum_{i=1}^{d_c} \oplus u_{n_i}) = L(u_{n_i} \oplus \sum_{j=1, j\neq i}^{d_c} \oplus u_{n_j}) \\ &= \log \frac{1+e^{L(\sum_{j=1,\, j\neq i}^{d_c} \oplus u_{n_j}) + L(u_{n_i})}}{e^{L(\sum_{j=1,\, j\neq i}^{d_c} \oplus u_{n_j})} + e^{L(u_{n_i})}}. \end{aligned} \tag{9}$$

Note that the term $L(\sum_{j=1, j\neq i}^{d_c} \oplus u_{n_j})$ is exactly equivalent to the outgoing message $\Lambda_{m\to n_i}(u_{n_i})$ from check node m to all the symbol nodes $u_{n_i} \in \{u_{n_1}, u_{n_2}, \ldots, u_{n_{d_c}}\}$, while $L(u_{n_i})$ is the incoming messag $\lambda_{n_i\to m}(u_{n_i})$. Thus (9) becomes

$$L(S_m) = \log \frac{1+e^{\Lambda_{m\to n_i}(u_{n_i}) + \lambda_{n_i\to m}(u_{n_i})}}{e^{\Lambda_{m\to n_i}(u_{n_i})} + e^{\lambda_{n_i\to m}(u_{n_i})}}.$$

After some algebra, we finally obtain

$$\Lambda_{m\to n_i}(u_{n_i}) = \log \frac{e^{\lambda_{n_i\to m}(u_{n_i}) + L(S_m)} - 1}{e^{\lambda_{n_i\to m}(u_{n_i}) - L(S_m)} - 1} - L(S_m). \tag{10}$$

We define

$$\Lambda_{m\to n_i}(u_{n_i}) \stackrel{def}{=} L(u_{n_i} \ominus S_m), \quad i = 1, \ldots, d_c. \tag{11}$$

Clearly, for each $i \in \{1, 2, \ldots, d_c\}$, the extrinsic information $\Lambda_{m \to n_i}(u_{n_i})$ can be computed simultaneously by a parallel implementation of the new core operation $L(u_{n_i} \ominus S_m)$ as shown in Fig. 3. Clearly, only $d_c - 1$ core operations of type $L(U \oplus V)$ and d_c core operations of type $L(U \ominus V)$ are necessary for a particular check-node update in this parallel topology.

Observe that (10) can be written as

$$\begin{aligned} \Lambda_{m \to n_i}(u_{n_i}) &= \log|e^{\lambda_{n_i \to m}(u_{n_i}) + L(S_m)} - 1| \\ &\quad - \log|e^{\lambda_{n_i \to m}(u_{n_i}) - L(S_m)} - 1| \\ &\quad - L(S_m). \end{aligned} \tag{12}$$

In (12) the calculation of the function $h(x) = \log|e^x - 1|$ is required, whose plot is given in Fig. 4. As can be seen in Fig. 4, the function $h(x)$ approaches $-\infty$ as x approaches zero. This behavior makes it difficult to use a look-up table with a small number of quantization levels for implementing the new core operation $L(U \ominus V)$. On the other hand, $h(x)$ can easily be approximated by a piece-wise linear function where the multiplying factors are powers of two and therefore simple to implement in hardware with shift operations. Table III is a very accurate piece-wise linear approximation of $h(x)$ with only eight regions. Note that such a piece-wise linear approximation is similar in implementation complexity to a 3-bit (eight quantization levels) table look-up. In summary, each $L(U \ominus V)$ takes four additions and two linear function evaluations.

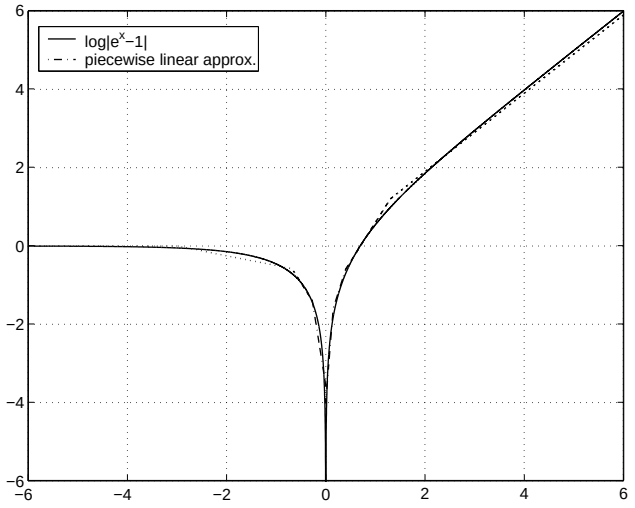

Fig. 4. The function $h(x) = \log|e^x - 1|$.

TABLE III

PIECEWISE LINEAR FUNCTION APPROXIMATION FOR $h(x) = \log|e^x - 1|$.

x	$\log\|e^x - 1\|$	x	$\log\|e^x - 1\|$
[-∞,-3)	0	[0, 0.15)	$2^4x - 4$ 0
[-3,-0.68)	$-2^{-2}x - 0.75$	[0.15, 0.4)	$2^2x - 2.2$
[-0.68,-0.27)	$-2x - 1.94$	[0.4, 1.3)	$2x - 1.4$
[-0.27, 0.0)	$-2^3x - 3.56$	[1.3, +∞)	$x - 0.1$

V. SIMULATION RESULTS

Simulation results are presented for the following LDPC decoding algorithms: the SPA, the LLR-SPA using the trellis topology for the check-node updates (designated as "LLR-SPA1"), and the LLR-SPA using the tree topology for the check-node updates (designated "LLR-SPA2"). Furthermore, the correction term $\log(1 + e^{-|L(U)+L(V)|}) - \log(1 + e^{-|L(U)-L(V)|})$ in the core operation $L(U \oplus V)$ of the LLR-SPA1 has been computed using either the look-up table shown in Table I or the piece-wise linear function shown in Table II. In addition, further simplifactions of the LLR-SPA1 have been simulated in which the correction term in the core operation $L(U \oplus V)$ is approximated by a fixed constant or eliminated entirely. The last case corresponds to the afore-mentioned sign-min approximation. Finally, the core operation involved in LLR-SPA2 is implemented using the piece-wise linear function shown in Table III. The results are obtained via Monte Carlo simulations in which the maximum number of iterations is fixed to 80 in all cases.

Figs. 5 and 6 show the bit error rate performance of an $[N = 1008, K = 504]$ LDPC code from [11] and an $[N = 6000, K = 3000]$ randomly constructed LDPC code, respectively, assuming an AWGN channel. For both codes, we observe that at a bit error rate of 10^{-5}, the simple sign-min approximation suffers a performance penalty of 0.3 to 0.5 dB. It appears that the loss in performance is greater as the number of parity-check equations of the LDPC code increases. On the other hand, all other reduced-complexity variants of the LLR-SPA perform very close to the conventional SPA. In particular, the piece-wise linear approximations of the core operations in LLR-SPA1 or LLR-SPA2 appear to suffer no loss (essentially less than 0.05 dB) in performance even in the case of the $[N = 6000, K = 3000]$ LDPC code, which involves 3000 parity-check equations. Furthermore, as can be seen in Fig. 5, the simple LLR-SPA1 algorithm that uses a constant correction term ($c = 0.8$) is also able to achieve the performance of the conventional SPA, in particular at higher SNRs.

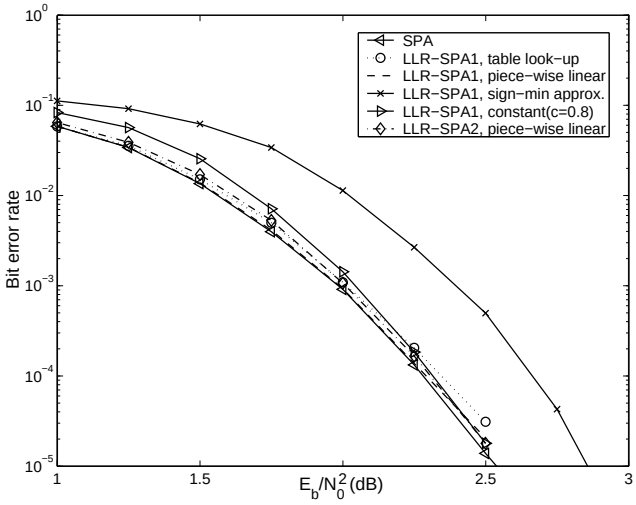

Fig. 5. Performance of $[N = 1008, K = 504]$ LDPC code from [11].

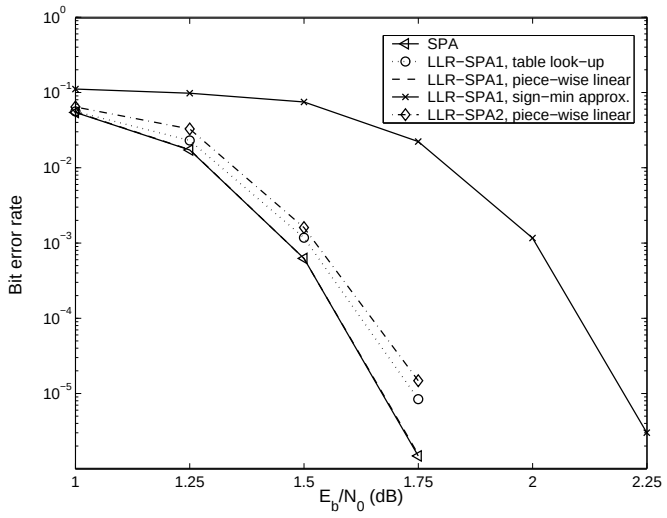

Fig. 6. Performance of $[N = 6000, K = 3000]$ LDPC code.

VI. Conclusions

Efficient implementations of the SPA for decoding LDPC codes have been considered. A number of reduced-complexity variants of the SPA based on using LLRs as messages between symbol nodes and check nodes have been investigated. In particular, two different topologies for implementing the check-node update, namely, trellis and tree topologies, have been presented. It was shown that the trellis topology would require $3(d_c - 2)$ core operations for the check-node update with a latency of the order $O(d_c)$. On the other hand, the tree topology requires $2(d_c - 1)$ core operations of the check-node update with a latency of the order $O(\log d_c)$.

The core operations are somewhat different in the two cases. Nevertheless, the correction terms in these core operations can be implemented via look-up tables or piece-wise linear functions, or even by using a single constant, facilitating simple hardware design. Simulations results have shown that it is possible to attain the performance of the conventional SPA extremely closely with a significant reduction in implementation complexity.

References

[1] R. G. Gallager, "Low-density parity-check code," *IRE Trans. Inform. Theory*, vol. IT-8, pp. 21-28, Jan. 1962.

[2] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Inform. Theory*, vol. 45, pp. 399-431, Mar. 1999.

[3] S.-Y. Chung, G. D. Forney, Jr., 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, pp. 58-60, Feb. 2001.

[4] L. Ping and W. K. Leung, "Decoding low density parity check codes with finite quantization bits," *IEEE Commun. Lett.*, vol. 4, pp. 62-64, Feb. 2000.

[5] M. P. C. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low density parity check codes based on belief propagation," *IEEE Trans. Commun.*, vol. 47, pp. 673-680, May 1999.

[6] E. Eleftheriou, T. Mittelholzer and A. Dholakia, "Reduced-complexity decoding algorithm for low-density parity-check codes," *IEE Electronics Letters*, vol. 37, pp. 102-104, Jan. 2001.

[7] V. Sorokine, F.R. Kschischang, and S. Pasupathy, "Gallager codes for CDMA applications – part II: Implementations, complexity, and system capacity," *IEEE Trans. Commun.*, vol. 48, pp. 1818-1828, Nov. 2000.

[8] J. Hagenauer, E. Offer, and L. Papke, "Iterative decoding of binary block and convolutional codes," *IEEE Trans. Inform. Theory*, vol. 42, pp. 429-445, Mar. 1996.

[9] A. J. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *IEEE J. Sel. Areas Commun.*, vol. 16, pp. 269-264, Feb. 1998.

[10] P. Robertson, E. Villebrun, and P. Hoeher, "A comparison of optimal and sub-optimal MAP decoding algorithms operating in the log domain," *Proc. Intl. Commun. Conf. '95*, pp. 1009-1013, June 1995.

[11] D.J.C. MacKay, "Online database of low-density parity-check codes," available at `http://wol.ra.phy.cam.uk/mackay/codes/data.html`.