Serial Sum-Product Architecture for Low-Density Parity-Check Codes

Conference Paper · September 2007

DOI: 10.1109/ICCCN.2007.4317812 · Source: IEEE Xplore

3 authors, including:

Ruwan Ratnayake
London Metropolitan University
7 PUBLICATIONS **21** CITATIONS

SEE PROFILE

Gu-Yeon Wei
Harvard University
234 PUBLICATIONS **7,697** CITATIONS

SEE PROFILE

Serial Sum-Product Architecture for Low-Density Parity-Check Codes

Ruwan N.S. Ratnayake
School of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138
Email: ratnayak@fas.harvard.edu

Erich F. Haratsch
LSI Corporation
Allentown, PA 18109

Gu-Yeon Wei
School of Engineering and Applied Sciences
Harvard University
Cambridge, MA 02138
Email: guyeon@eecs.harvard.edu

***Abstract*—A serial sum-product architecture for low-density parity-check (LDPC) codes is presented. In the proposed architecture, a standard bit node processing unit computes the bit to check node messages sequentially, while the check node computations are broken up into several steps and computed on the fly. This bit node centric architecture requires considerably less memory compared to other serial architectures, including the check node centric architecture.**

Index Terms - Low-density parity-check (LDPC) codes, sum-product algorithm, bi-partite graph, serial architecture.

I. Introduction

The spectacular success of turbo codes with its iterative decoding and message passing mechanism has inspired the rediscovery of low density parity-check (LDPC) codes [1], [2]. Though LDPC codes were first discovered by Gallager as early as in the 1960s [3] they were not considered for practical use due to their immense computational complexities. Recent work has shown that LDPC codes achieve near Shannon limit performance [4]. However until recently the lack of efficient implementation techniques has prevented LDPC codes from being considered for error correction in several applications.

One of the inherent difficulties in finding implementable architectures for LDPC decoder architectures is that LDPC codes have less structure than other codes such as trellis or turbo codes. An LDPC code is defined by its random like parity-check matrix which dictates the communications between the bit and check nodes. On the other hand other codes such as turbo codes have a more regular structure. The random nature in turbo codes is accomplished by an interleaver which is external to the code. Computations and message passing in turbo codes can be easily separated and the decoder primarily consists of message computation circuitry. However, randomness is inherent to the LDPC code and the message passing is defined by the code itself. Thus, the message computations and message passing in LDPC codes are inseparable, which complicates the implementation.

One implementation method for LDPC codes is a fully parallel architecture where the exact copy of the bi-partite graph of the code is built in hardware. The bit and check node computations are implemented as combinational logic and the edges of the graph are simply the wires that connect the appropriate nodes as defined by the graph. This method achieves high throughput since an entire block is processed simultaneously. However, the main drawback of this parallel architecture is the tremendous wiring complexity due to the node connections. Given that the number of messages passed along the edges in a typical LDPC code is extremely large, and given that each message needs to be represented with a number of bits large enough for an adequate resolution of the messages, the number of wires needed to connect the nodes are in the orders of tens of thousands even for a code of moderate length in the range of about one thousand bits. The large wiring complexity significantly increases the area and reduces the utilization. The LDPC decoder chip by Blanksby *et al.* [5] is implemented in the fully parallel architecture and has an area of about 53.5 mm^2 in 0.16μm technology and dedicates about half of this area for wire routing.

The serial architecture can overcome the area overhead by saving the messages in memory and processing bit node and check node computations sequentially. Several proposals for serial architectures have appeared in literature [6], [7]. Wu *et al.* [8] and Hocevar [9] have proposed serial architectures that process either the bit node or check node computations on the fly. The architecture proposed in this paper also performs some computations on the fly. However we will show that our method requires significantly less memory and/or less number of clock cycles to process a given block of code.

This paper is organized as follows. First, we give a brief overview of LDPC codes. Then in Section III we give an in-depth description of our proposed architecture. Subsection III-A describes the memory requirements and Subsection III-B compares the merits of the proposed architecture with other architectures found in literature. Finally, Section IV provides concluding remarks.

II. Low-Density Parity-Check Codes

An LDPC code is defined by a binary parity-check matrix ($\mathbf{H}$). Though the matrix size is typically large, it is sparse. The matrix has n columns and m rows as shown in (1). The

1-4244-1251-X/07/$25.00 2007 IEEE.

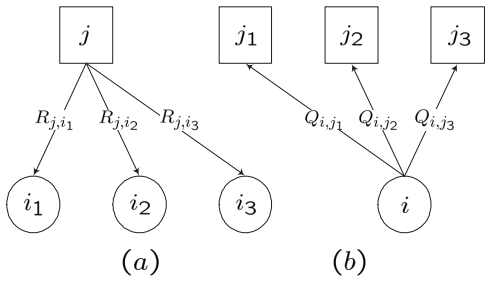

Fig. 1. Message passing (a) check to bit nodes, $\mathcal{C}_j = \{i_1, i_2, i_3\}$ (b) bit to check nodes, $\mathcal{B}_i = \{j_1, j_2, j_3\}$

information in the parity-check matrix can also be represented by a bi-partite graph. Each column and row corresponds to a bit and check node in the graph respectively. A bit node i is connected to a check node j by an edge if there is a 1 at the $(i,j)^{th}$ position in $\mathbf{H}$. For each bit node i, the set of participating check nodes, i.e., the position of 1s in the column i, is denoted by $\mathcal{B}_i$. Similarly the set of participating bit nodes for the check node j is denoted by $\mathcal{C}_j$. The number of 1s in a row is called the row degree and denoted by d_r and number of 1s in a column is called column degree and denoted by d_c. For simplicity this paper considers only regular codes, where the row and column degrees are constant.

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 1 & 0 & 0 & \cdots & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ 0 & 1 & 1 & \cdots & 0 \end{bmatrix} \tag{1}$$

Two types of messages are passed along the edges between the bit and check nodes. The messages passed from bit node i to check node j are denoted by $Q_{i,j}$, where $j \in \mathcal{B}_i$. Similarly the messages computed at the check node j and passed to the participating bit nodes are denoted $R_{j,i}$, where $i \in \mathcal{C}_j$. Fig. 1 shows the messages passed between the nodes.

An LDPC code can be decoded using the sum-product algorithm, which consists of four steps per iteration. In the following equations, the iteration index is denoted by k.

1) Initialization
   All the messages from check node j to bit node i are initialized to zero i.e. $R^0_{j,i} = 0$.
2) Bit node processing
   For each $i \in \{1, ..., n\}$ compute $Q^k_{i,j}$, where $j \in \mathcal{B}_i$

$$Q^k_{i,j} = \sum_{l \in \mathcal{B}_i, l \neq j} R^k_{l,i} + \lambda_i \tag{2}$$

   where λ_i is the *a priori* log-likelihood ratio of the bit i.

3) Check node processing
   For each $j \in \{1, ..., m\}$ compute $R^k_{j,i}$, where $i \in \mathcal{C}_j$

$$R^k_{j,i} = s^k_{j,i} \times \phi^{-1}\left(\sum_{l \in \mathcal{C}_j, l \neq i} \phi\left(|Q^k_{l,j}|\right)\right) \tag{3}$$

   where the sign bit $s^k_{j,i}$ is given by

$$s^k_{j,i} = \prod_{l \in \mathcal{C}_j, l \neq i} \text{sign}(Q^k_{l,j}) \tag{4}$$

   and

$$\phi(x) = -\log\left(\tanh\frac{x}{2}\right) = \log\left(\frac{e^x+1}{e^x-1}\right) = \phi(x)^{-1} \tag{5}$$

4) *A posteriori* log-likelihood ratio (LLR) computation
   After K iterations, the LLR for each bit i is computed by

$$\Lambda_i = \sum_{l \in \mathcal{B}_i} R^K_{l,i} + \lambda_i \tag{6}$$

The equations (2) - (6) are well known and cited here for reference.

III. Proposed Serial Sum-product Architecture for LDPC Decoder

This paper proposes an architecture that performs check node computations on the fly. This is achieved by breaking the check node computation (3) up into several steps. Consider the following definitions:

$$\rho^k_{i,j} = \phi(|Q^k_{i,j}|) \tag{7}$$

$$\sigma^k_{i,j} = \text{sign}(Q^k_{i,j}) \tag{8}$$

$$P^k_j = \sum_{l \in \mathcal{C}_j} \rho^k_{l,j} \qquad j \in \{1, ..., m\} \tag{9}$$

$$S^k_j = \prod_{l \in \mathcal{C}_j} \sigma^k_{l,j} \qquad j \in \{1, ..., m\} \tag{10}$$

where $\rho^k_{i,j}$ is the transformed magnitude of the bit to check node message $Q_{i,j}$ and $|\,|$ is the notation for magnitude. Based on these variables, the sign and magnitude of the message from check node j to bit node i is given by:

$$|R^k_{j,i}| = \phi(P^k_j - \rho^k_{i,j}) \tag{11}$$

$$\text{sign}(R^k_{j,i}) = S^k_j \times \sigma^k_{i,j} \tag{12}$$

The check node computations are performed in sign-magnitude format, while the bit node computations can be performed in 2's-complement format.

A block diagram of the proposed architecture is shown in Fig. 2. The illustration is for a simplified example where $d_c = 3$, $\mathcal{B}_i = \{j1, j2, j3\}$. At each time cycle the bit node processing unit (BNU) performs computations pertaining to one bit node. This is called bit node centric architecture. It takes n cycles to perform the entire computations for bit nodes 1 to n by using one bit node processing unit. Within the first time cycle, the first bit node $(i = 1)$ is computed. Then at $(n(k-1)+i)^{th}$ time cycle, the decoder is computing the i^{th} bit node at k^{th} iteration and produces messages $Q^k_{i,j}, \forall j \in \mathcal{B}_i$.

The bit to check node messages are converted from 2's-complement to sign-magnitude format. There are d_c number

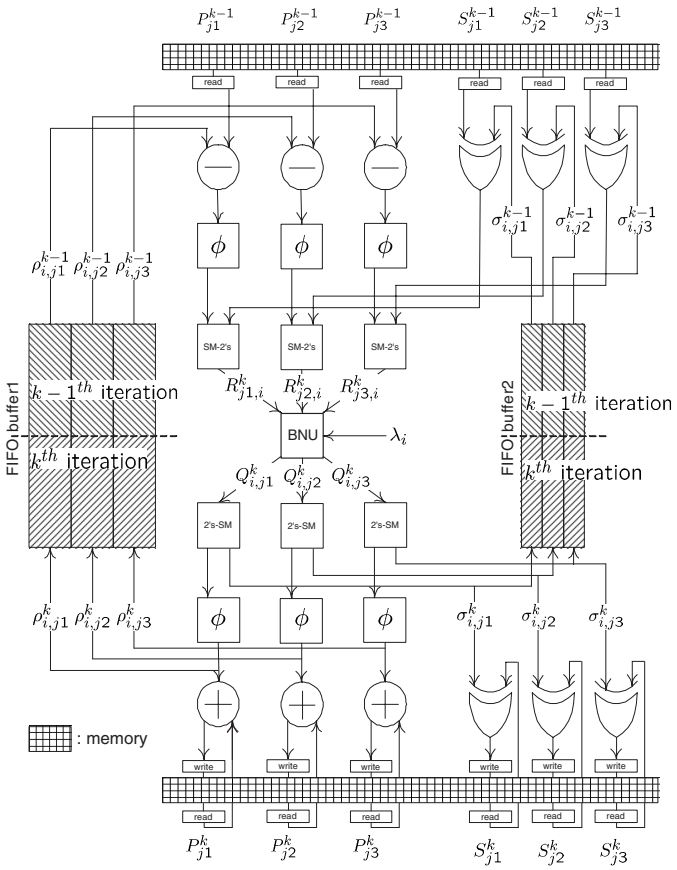

Fig. 2. Block diagram of the proposed architecture for $d_c = 3, \mathcal{B}_i = \{j1, j2, j3\}$

of messages generated at each cycle. The magnitude values of the messages are sent to combinational logic units that perform the function ϕ and the outputs from these units are the transformed magnitudes, $\rho^k_{i,j}, j \in \mathcal{B}_i$ as defined in (7). These transformed magnitudes are fed to corresponding update units.

The update units consist of adders and memory access circuitry. These update units can access any d_c number of memory elements from m memory elements (a memory element consists of a specified number of memory bits). The update units update the relevant memory elements such that at the end of an iteration (i.e. nk^{th} time cycle) the values $P^k_j, \forall j \in \{1...m\}$ as defined in (9) are stored in the m memory elements. In other words these memory elements keep a running sum of the ρ's. If the intermediate value in the memory element $j \in \{1...m\}$ for the k^{th} iteration is given by $P^k_j(i)$ for $i \in \{1...n\}$, then all the running sums saved in the memory at the $(n(k-1)+i)^{th}$ time instance are given by:

$$P^k_j(i) = \begin{cases} P^k_j(i-1) + \rho^k_{i,j} & \text{if } j \in \mathcal{B}_i \\ P^k_j(i-1) & \text{else} \end{cases} \quad (13)$$

Then at the end of an iteration the memory contains:

$$P^k_j = P^k_j(n) \quad (14)$$

The sign values, $\sigma^k_{i,j}, j \in \mathcal{B}_i$ as defined by (8) are processed in a similar manner. The sign values are fed into another set of update units consisting of XOR and memory access circuitry. These update units update the relevant memory bits such that at the end of an iteration the memory bits contain $S^k_j, j \in \{1...m\}$ as defined in (10). In other words these memory bits keep a running product of the sign-bit σ's. The sign product is obtained by XOR gates. The running sign product stored in the memory is given by:

$$S^k_j(i) = \begin{cases} S^k_j(i-1) \times \sigma^k_{i,j} & \text{if } j \in \mathcal{B}_i \\ S^k_j(i-1) & \text{else} \end{cases} \quad (15)$$

where $S^k_j(i)$ is the intermediate value for the memory bit $j \in \{1...m\}$ at the k^{th} iteration. At the end of an iteration the memory contains:

$$S^k_j = S^k_j(n) \quad (16)$$

The transformed magnitudes and sign values are also fed into the two first-in first-out (FIFO) buffers buffer 1 and 2 in Fig. 2 respectively. Without loss of generality it is assumed that each message is represented by q bits, where the magnitude and transformed magnitude consist of $q-1$ bits and the sign comprises 1 bit. Since d_c number of transformed magnitudes and sign values are computed at each time cycle, the buffers 1 and 2 require $d_c(q-1)$ and d_c bits per row respectively. At each cycle the messages fill in one row of each buffer. Since the decoder saves information pertaining to the entire code block, the buffers have n such rows.

At time cycle $n(k-1)+i$, the set of $\rho^k_{i,j}, j \in \mathcal{B}_i$ is fed to the bottom of buffer 1 and the set of $\rho^{k-1}_{i,j}, j \in \mathcal{B}_i$ from the previous iteration is read from the top of buffer 1. Similarly within this cycle, the sign values $\sigma^k_{i,j}, j \in \mathcal{B}_i$ are fed to the bottom of buffer 2 and the sign values from the previous iteration $\sigma^{k-1}_{i,j}, j \in \mathcal{B}_i$ are read from the top.

After completing n such cycles, i.e. at nk^{th} time cycle, the buffers 1 and 2 contain $\rho^k_{i,j}$ and $\sigma^k_{i,j}, \forall i \in \{1...n\}, j \in \mathcal{B}_i$ respectively. Each row i in the buffers contains the information pertaining to bit node i. The bottom row of buffer 1 contains $\rho^k_{1,j}, j \in B_1$, and the top row contains $\rho^k_{n,j}, j \in B_n$. Similarly the rows in buffer 2 contain $\sigma^k_{1,j}, j \in B_1$ to $\sigma^k_{n,j}, j \in B_n$ consecutively in this order. At this time cycle the updated memory contains the corresponding magnitudes P^k_j and signs $S^k_j, \forall j \in \{1...m\}$.

The check to bit node messages $R^k_{j,i}$ are computed based on the values computed in the previous iteration, namely P^{k-1}_j and S^{k-1}_j, $j \in \{1...m\}$ saved in the memory and $\rho^{k-1}_{i,j}$, $\sigma^{k-1}_{i,j}, j \in \mathcal{B}_i$ saved in the FIFO buffers 1 and 2. At each iteration, P^{k-1}_j and $\rho^{k-1}_{i,j}, j \in \mathcal{B}_i$ pertaining to bit node i are read from the memory and FIFO buffer 1 respectively. The relevant intermediate results $(P^{k-1}_j - \rho^{k-1}_{i,j})$ are computed for each $j \in \mathcal{B}_i$ and passed to combinational units that perform the function ϕ. The outputs of these functional units are the magnitudes of the messages from the corresponding

TABLE I
COMPARISON OF ARCHITECTURES

Architecture	Memory	Clock Cycles
Standard	$2nd_cq$	$n+m$
Check node centric	$(2n+md_r)q$	m
Bit node centric	$(2m+nd_c)q$	n

check nodes to the i^{th} bit node at k^{th} iteration, namely $|R^k_{j,i}|, j \in \mathcal{B}_i$ according to (11).

The sign value of the check to bit nodes is evaluated in a similar manner. At k^{th} iteration, the required S^{k-1}_j and $\sigma^{k-1}_{i,j}, j \in \mathcal{B}_i$ are read from the memory and the buffer 2 respectively. The product $(S^{k-1}_j \times \sigma^{k-1}_{i,j})$ is computed using XOR gates for each $j \in \mathcal{B}_i$. These are the sign values of the corresponding messages from the d_c check nodes to the i^{th} bit node at k^{th} iteration, namely $\text{sign}(R^k_{j,i}), j \in \mathcal{B}_i$ according to (12). The sign and magnitude values are passed through combinational logic that converts from the sign-magnitude to 2's-complement format. The results $R^k_{j,i}, j \in \mathcal{B}_i$ are the inputs to the bit node processing unit for the i^{th} bit node at k^{th} iteration.

A. Memory Requirement and Throughput

FIFO buffers 1 and 2 require $nd_c(q-1)$ and nd_c memory bits respectively. $2(q-1)m$ memory bits are required to store $P_j, \forall j \in \{1...m\}$. The multiplication factor 2 is due to the need to save the current computations pertaining to iteration k while simultaneously reading the values from the previous iteration $k-1$. Similarly, saving all $S_j, \forall j \in \{1...m\}$ requires $2m$ memory bits. Thus the decoder requires a total of $(2m+nd_c)q$ memory bits.

At each time cycle the proposed method computes d_c number of check to bit node messages and d_c number of bit to check node messages. Thus it takes n cycles to process a data block of length n per iteration using one bit node processing unit.

B. Advantages Over Other Methods

The proposed architecture performs the sum-product algorithm as defined by the equations (7) to (12). This method does not lose information such as the method proposed in [8], where only the minimum values are considered. More over the method described in [8] is for a concatenated detector-decoder system only.

Hocevar describes an architecture in [9] that also computes both bit to check and check to bit messages within one cycle. However Hocevar's architecture is check node centric since it uses standard check node computational units and breaks up the bit node computations. Mansour *et al.* [7] also describe a method that is very similar to the architecture in [9]. The check node centric method described in [9] requires $(2n+md_r)q$ memory bits to save the required messages and intermediate values. On the other hand, our proposed bit node centric method requires $(2m+nd_c)q$ memory bits. The difference in memory requirements is $2(n-m)q$, since $md_r == nd_c$. Thus the bit node centric architecture always requires less memory than the check node centric one. The memory savings are particulary significant for high rate codes, which are desirable for example in magnetic storage. Rates as high as $\frac{8}{9}$ are typical in magnetic storage. Assuming $d_c = 3$ (which in turn dictates $d_r = 27$ for regular codes with rate $\frac{8}{9}$) the check node centric method needs $45mq$ memory bits whereas the proposed method needs only $29mq$ bits. Thus in this typical scenario, the check node centric architecture requires close to 50% more memory compared to the bit node centric method.

Moreover, a check node processing unit needs to sum d_r number of messages whereas a bit node processing unit only needs to sum up d_c messages. For high rate codes d_r is considerably larger than d_c. Thus, for such codes the check node centric architecture would need a significantly larger adder tree than the bit node centric architecture.

In terms of number of cycles required for each iteration, the check node centric method requires m cycles if it uses one check node processing unit. The bit node centric method requires n cycles assuming it uses one bit node processing unit. However, the cycle periods for both methods are significantly different. Minimum cycle periods are determined by the critical path delay. The critical path for the check node centric method is greater than for the bit node centric method (one reason is the larger adder tree). Both methods need to perform the same number of computations to obtain the relevant messages. Thus it takes a comparable amount of time for one iteration (though the number of cycles are different) for both methods if the same number of bit node or check node processing units are assumed. Thus, the proposed bit node centric method requires n number of shorter cycles whereas the check node centric method requires m number of longer cycles per iteration, giving comparable throughput.

Table I summarizes the memory requirements and number of clock cycles needed for different architectures. For a fair comparison the standard architecture considered has one bit node and one check node processing unit. This scheme switches read and write operations between two memory banks. At each cycle it reads messages from one memory bank and writes the results to the other.

IV. CONCLUSION

This paper presents a serial architecture for an LDPC decoder that performs the sum-product algorithm. This architecture uses a standard bit node processing unit, while the check node computations are broken up and computed on the fly. This bit node centric architecture requires significantly less memory than other serial architectures, especially for codes with high code rate.

REFERENCES

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

[2] T. J. Richardson and R. L. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Info. Theory*, vol. 47, pp. 599–618, Feb. 2001.

[3] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Info. Theory*, pp. 21–28, 1962.

[4] S. Y. Chang, G. D. Forney, T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 db of the Shannon limit," *IEEE Comm. Lett.*, vol. 5, pp. 58–60, Feb. 2001.

[5] A. Blanksby and C. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE J. Solid-State Circuits*, vol. 37, pp. 404–412, Mar. 2002.

[6] E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam, "VLSI architectures for iterative decoders in magnetic recording channels," *IEEE Trans. Mag.*, vol. 37, pp. 748–755, Mar. 2001.

[7] M. M. Mansour and N. R. Shanbhag, "High-throughput LDPC decoders," *IEEE Trans. VLSI Systems*, vol. 11, pp. 976–996, Dec. 2003.

[8] Z. Wu and G. Burd, "Equation based LDPC decoder for intersymbol interference channels," *IEEE Int'l Conf. on Acoustics, Speech, and Signal Processing (ICASSP)*, vol. 5, pp. 757– 760, Mar. 2005.

[9] D. E. Hocevar, "LDPC code construction with flexible hardware implementation," *IEEE Int'l Conf. on Comm. (ICC)*, vol. 4, pp. 2708–2712, May 2003.

1-4244-1251-X/07/$25.00 2007 IEEE.