EFFICIENT DSP IMPLEMENTATION OF AN LDPC DECODER

Gottfried Lechner, Jossy Sayir

Telecommunications Research
Center Vienna (ftw.)
A-1220 Vienna, Austria
{lechner,sayir}@ftw.at

Markus Rupp

TU Vienna, Institute for
Communication and RF Engineering
A-1040 Vienna, Austria
mrupp@nt.tuwien.ac.at

ABSTRACT

We present a high performance implementation of a belief propagation decoder for decoding low-density parity-check (LDPC) codes on a fixed point digital signal processor. A simplified decoding algorithm was used and a stopping criteria for the iterative decoder was implemented to reduce the average number of required iterations. This leads to an implementation with increased throughput compared to other implementations of LDPC codes or Turbo codes. This decoder is able to decode at 5.4Mbps on a Texas Instruments TMS320C64xx DSP running at 600MHz.

1. INTRODUCTION

Low-density parity-check (LDPC) codes were introduced by Gallager [1] in 1962. They allow transmission at rates close to channel capacity while having a decoding complexity which is linear in the block length. Due to lack of powerful processors at the time of their invention, they were mostly forgotten, then rediscovered after the invention of Turbo codes [2].

In this paper, we present an implementation of an LDPC decoder on a digital signal processor (DSP). The DSP used is a Texas Instruments TMS320C64xx fixed point processor. We present an efficient implementation utilizing the parallel units of this DSP. Furthermore, we compare the data throughput with existing DSP implementations of Turbo codes.

The rest of this paper is organized as follows: in Section 2, we describe LDPC codes, the decoding algorithm and an approximation of this algorithm that is suited for fixed point operations. The implementation on the DSP is described in Section 3. In Section 4, a stopping criterion is introduced and Section 5 shows the computation requirements of the implementation. In Section 6, simulation results are presented and compared with an existing implementation of a Turbo decoder.

2. LDPC CODES

LDPC codes are block codes described by a parity-check matrix. The term low-density originates from the fact that the number of ones in the parity-check matrix is small compared to the block length. Therefore, the matrix is sparse, which leads to a decoding algorithm with a computational complexity that is linear in the block length [3].

This decoding algorithm works iteratively by passing messages on the edges of the associated factor graph, which is a bipartite graph containing variable nodes (representing the digits of the codeword) and check nodes (representing the parity-check equations). The messages are Log-Likelihood-Ratios (LLR), i.e. the sign represents the binary digit ($0 \rightarrow +$, $1 \rightarrow -$) and the magnitude represents the reliability of the decision.

This paper focuses on regular codes, i.e. every variable node is connected to d_v check nodes and every check node is connected to d_c variable nodes. A factor graph of a code with $d_v = 3$ and $d_c = 6$ is shown in Figure 1. The connections between the variable nodes and the check nodes are realized using an edge interleaver.

The operations at the variable and check nodes are described in the next subsections following the notation of [4]. $Z_{mn}^{(i)}$ denotes a message sent from variable node n to check node m at iteration i and $L_{mn}^{(i)}$ denotes a message sent from check node m to variable node n at iteration i. The set of neighboring check nodes of a variable node n is denoted as $\mathcal{M}(n)$ and the set of neighboring variable nodes of a check node m is denoted as $\mathcal{N}(m)$.

2.1. Variable Nodes

Each variable node receives one message $L_n^{(0)}$ from the channel and one message $L_{mn}^{(i)}$ from every check node it is connected to. In every iteration, the variable node has to calculate an outgoing message $Z_{mn}^{(i)}$ to every con-

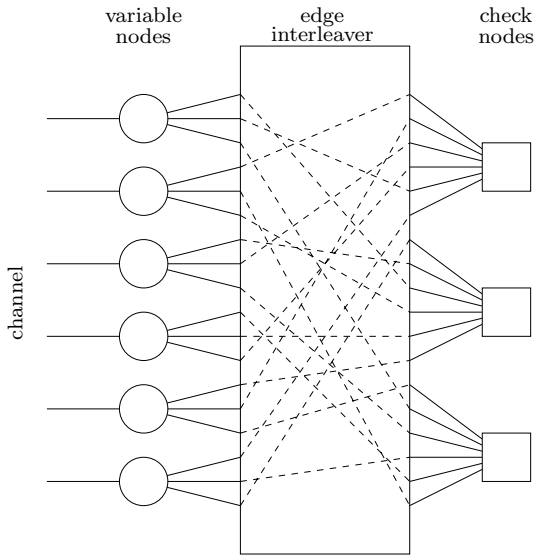

Fig. 1. Factor Graph of a LDPC Code.

nected check node according to

$$Z_{mn}^{(i)} = L_n^{(0)} + \sum_{m' \in \mathcal{M}(n) \setminus m} L_{m'n}^{(i)}. \quad (1)$$

After every iteration, the variable nodes have to calculate an estimate of the a-posteriori LLRs

$$A_n^{(i)} = L_n^{(0)} + \sum_{m' \in \mathcal{M}(n)} L_{m'n}^{(i)}. \quad (2)$$

These operations are already suited for implementation on a fixed point DSP and therefore, no simplifications are made.

2.2. Check Nodes

Every check node represents a single parity-check code. The outgoing messages $L_{mn}^{(i)}$ of a check node can be computed as

$$L_{mn}^{(i)} = 2 \operatorname{atanh} \left[\prod_{n' \in \mathcal{N}(m) \setminus n} \tanh \frac{Z_{mn'}^{(i-1)}}{2} \right]. \quad (3)$$

This computation can not be implemented simply on a fixed point DSP. Therefore, we use an approximation similar to the Max-Log-MAP algorithm

$$L_{mn}^{(i)} \approx \min_{n' \in \mathcal{N}(m) \setminus n} |Z_{mn'}^{(i-1)}| \cdot \prod_{n' \in \mathcal{N}(m) \setminus n} \operatorname{sgn}(Z_{mn'}^{(i-1)}). \quad (4)$$

To improve the performance, we also included a correction term as shown in [4] called *offset belief propagation based* decoding

$$L_{mn}^{(i)} \leftarrow \operatorname{sgn}(L_{mn}^{(i)}) \cdot \max(|L_{mn}^{(i)}| - \beta, 0). \quad (5)$$

Note that the correction term is constant and can be implemented with low complexity. For our implementation we used the results from [4], where the value of β is optimized when designing the code using density evolution.

3. IMPLEMENTATION

We implemented a regular LDPC code with $d_v = 3$, $d_c = 6$ ($rate = 0.5$) and a random edge interleaver in C extended by intrinsic functions for the C64. All computations in our implementation are performed with 16 bit quantization, a value large enough so that the loss in performance due to quantization effects can be neglected. The implementation of the three parts of the decoder is described in the next sections.

3.1. Variable Nodes

The computation of the outgoing messages and the a-posteriori LLRs of a variable node can be done efficiently by computing first equation (2) and then subtracting the incoming message from the sum for every outgoing message

$$Z_{mn}^{(i)} = A_n^{(i)} - L_{mn}^{(i)}. \quad (6)$$

This leads to three additions and three subtractions per variable node.

3.2. Check Nodes

For an outgoing message of a check node, the product of the signs of all other incoming messages has to be computed. This can be done by first computing the product of the signs of all inputs and then multiplying every outgoing message with the sign of the associated incoming message.

The min operation of equation (4) can be efficiently calculated by splitting it up in the following way, where a_i and b_i represents the magnitude of an incoming and outgoing message respectively.

$$\begin{aligned} x_{12} &= \min(a_1, a_2) \\ x_{34} &= \min(a_3, a_4) \\ x_{56} &= \min(a_5, a_6) \\ x_{1234} &= \min(x_{12}, x_{34}) \\ x_{1256} &= \min(x_{12}, x_{56}) \\ x_{3456} &= \min(x_{34}, x_{56}) \end{aligned} \quad (7)$$

and calculating the outgoing messages as

$$\begin{aligned} b_1 &= \min(a_2, x_{3456}) \\ b_2 &= \min(a_1, x_{3456}) \\ b_3 &= \min(a_4, x_{1256}) \\ b_4 &= \min(a_3, x_{1256}) \\ b_5 &= \min(a_6, x_{1234}) \\ b_6 &= \min(a_5, x_{1234}) \end{aligned} \quad (8)$$

For every check node, we need six abs operations and six comparisons for computing the magnitude and sign of the incoming messages. Furthermore, we need 12 min operations, six additions and 24 logical operations for calculating the outgoing messages. All these operations can be implemented without using branches and many operations are independent from each other, allowing the compiler to optimize the code and to utilize the parallel computation units of this DSP.

3.3. Edge Interleaver

The interleaver is theoretically the easiest part of the implementation. However, it requires random like memory access operations which are very time consuming. We integrated the interleaver operations in the check nodes because there, it is possible to access the memory while computing the complex (in comparison to the variable nodes) outgoing messages of the check nodes.

All operations needed in our implementation (abs, min, max, add, sub and comp) are available as intrinsic functions that can interpret a 32 bit word as two 16 bit words and perform two operations in one instruction. To utilize this feature, we have to arrange the messages in memory in a special way so that messages from odd nodes are stored in the lower part and messages from even nodes are stored in the higher part of a 32 bit word.

This reordering of messages compared to Figure 1 can theoretically be done by placing an additional interleaver between the edge interleaver and the variable/check nodes. In practice, these two additional interleavers are included in the edge interleaver.

With this arrangement, we are able to calculate two variable nodes and two check nodes at the same time, i.e. doubling the data throughput.

4. STOPPING CRITERION

An advantage of the LDPC decoder in comparison to a Turbo decoder is that the decoder can easily detect whether it reached a valid codeword. This is done by verifying that every check node is fulfilled, i.e. the product of the signs of the incoming messages of a check node is positive for all check nodes. Since we calculated the product of the signs already we can use it to stop iterating when the decoder reached a valid codeword. Otherwise, we will continue iterating until a predefined maximum number of iterations is reached.

An additional gain of this stopping criterion is that the decoder is able to mark a block that has not been decoded successfully. This can be used by higher layers.

The implementation of the stopping criterion leads to a variable decoding time. An advantage of this property is that the average number of iterations can be used to estimate the E_b/N_0 of the channel if this is required.

Table 1. Computation Requirements of the LDPC Decoder.

	variable node	check node	correction
abs		6	
min/max		12	6
logical		24	
add/sub	6	6	6
comp		6	
load	4	18	
store	4	12	

5. IMPLEMENTATION REQUIREMENTS

The decoder was implemented in C and the number of cycles required per iteration was simulated. Including the correction term and the stopping criteria, we achieved a fixed computation time of 11.1 cycles/iteration/bit. For a DSP running at a clock frequency of 600MHz and an average number of 10 iterations, this results in a data throughput of 5.4Mbps. This implementation is approximately 2.6 times faster than a comparable Turbo Decoder implemented on the same type of DSP [5]. However, the Turbo decoder needs fewer iterations to converge. This makes the comparison between LDPC decoder and Turbo decoder difficult.

The computation requirements of our implementation are summarized in Table 1. The values given in the table do not account for the parallel computation of two nodes described in Subsection 3.3. Furthermore, when calculating the total number of operations for one iteration, it must be considered that the number of check nodes is half the number of variable nodes.

6. SIMULATION RESULTS

We simulated the bit error rate of the implemented regular LDPC decoder using a codeword length of $N = 10228$ and compared the performance with a Turbo decoder using an 8-state component encoder and an even-odd interleaver described in [5] with the same block length. The Turbo code was punctured by transmitting the two parity sequences alternatively to achieve the same rate as the LDPC code.

Figure 2 shows the bit error rate of both decoders for a given maximum number of iterations using binary phase shift keying and an additive white Gaussian noise channel.

To compare the required computation time, we show the number of cycles per bit in Figure 3. Due to the stopping criterion, the average number of cycles depends on E_b/N_0 for our decoder while the computation time of the Turbo decoder is constant. For low E_b/N_0 the computation time for the LDPC decoder is higher than for the Turbo decoder. For high E_b/N_0 the decoder is able to converge to the transmitted codeword after a few iterations which leads to a computation time that is lower than for a Turbo decoder.

7. CONCLUSION

In this paper, we showed how to implement a decoder for LDPC codes using a fixed point DSP. By using a correction term and an easy to implement stopping criterion, we were able to achieve similar performance as with a Turbo decoder. At high E_b/N_0, our LDPC decoder requires less computation time than the Turbo decoder. In addition, the LDPC decoder has the advantage that it is able to detect a unsuccessful decoding operation and can pass this information to higher layers.

Our implementation can be extended to different rates and to irregular LDPC codes. Irregular codes have a lower bit error rate for a given E_b/N_0 but usually require a higher number of iterations to converge. Further work should be done to design irregular LDPC codes with good convergence behavior and good performance in terms of bit error rate.

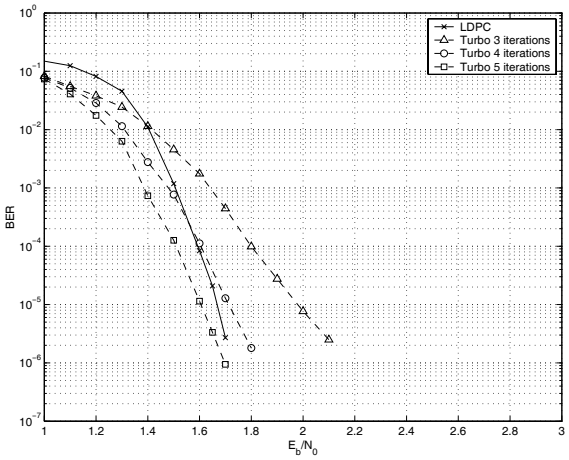

Fig. 2. Bit error rate.

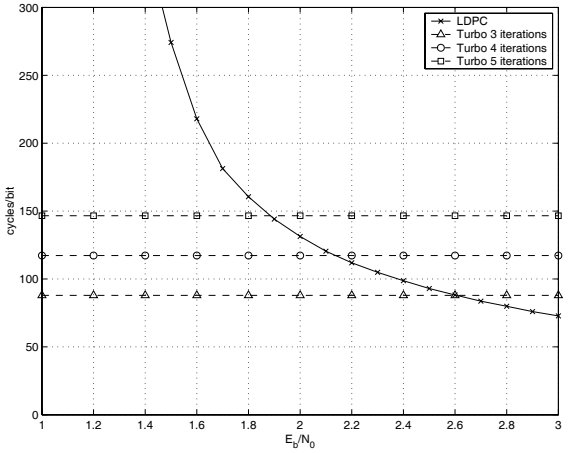

Fig. 3. Required cycles per bit.

8. REFERENCES

[1] R.G. Gallager, "Low density parity check codes," *IRE Transactions on Information Theory*, vol. IT-8, pp. 21–28, Jan 1962.

[2] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes," in *Proc. 1993 IEEE International Conference on Communications, Geneva, Switzerland*, 1993, pp. 1064–1070.

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

[4] Jinghu Chen and Marc P. C. Fossorier, "Density evolution for two improved BP-based decoding algorithms of LDPC codes," *IEEE Communications Letters*, vol. 6, no. 5, pp. 208–210, 2002.

[5] Yingtao Jiang, Yiyan Tang, Yuke Wang, and Dian Zhou, "A DSP-based turbo codec for 3G communication systems," in *Proc. 2002 IEEE International Conference on Acoustics, Speech, and Signal Processing*, 2002, pp. III–2685–III–2688.