PERFORMANCE ANALYSIS OF IEEE DEFINED LDPC CODES UNDER VARIOUS DECODING ALGORITHMS AND THEIR IMPLEMENTATION ON A RECONFIGURABLE INSTRUCTION CELL ARCHITECTURE

Zahid Khan, Tughrul Arslan, Ahmet T. Erdogan, Sami Khawam, Ioannis Nousias, Mark Milward, Ying Yi

System Level Integration Group
School of Engineering and Electronics,
The University of Edinburgh, Scotland, UK
z.khan@ed.ac.uk, Tughrul.Arslan@ee.ed.ac.uk

ABSTRACT

This paper builds a real time Programmable LDPC Decoder for decoding codes specified in IEEE 802.16 standard and discusses their performance under various decoding algorithms. Out of the decoding algorithms, the modified Min-Sum SPA is selected for implementation and optimization on a reconfigurable instruction cell architecture. Different general and architecture specific optimization techniques are applied to enhance the throughput. With the architecture, a throughput of 20 Mbps has been achieved.

I. INTRODUCTION

In this paper, we present implementation of a real time programmable LDPC decoder that can support IEEE defined LDPC codes [1]. Each code rate and code length is supported by different parity check matrices that are computed in real time from the model matrices stored in the memory. The proposed architecture can be implemented in ASIC, FPGA or DSP. We implemented it on a reconfigurable instruction cell architecture [2]. This architecture belongs to the emerging field of Reconfigurable Computing and is an effort to combine the flexibility and programmability of DSP, performance of FPGA and low power consumption of ASIC in one unified core so that the core can meet the requirement of next generation mobile systems.

Variable Node Processors: Each variable node in LDPC decoder receives one message $LLR_v^{(0)}$ from the channel and one message LLR_{cv}^i for the corresponding check nodes to which it is connected. In every iteration, a variable node has to calculate two messages: one Z_{vc}^i for each check node and the other a-posteriory LLR estimate

$$Z_{vc}^i = LLR_v^0 + \sum_{c'\varepsilon C(v)\backslash c} LLR_{c'v}^i \tag{1}$$

A_v^i for the bit in the frame which it represents. Here *C(v)* is the set of neighboring check nodes of variable node v.

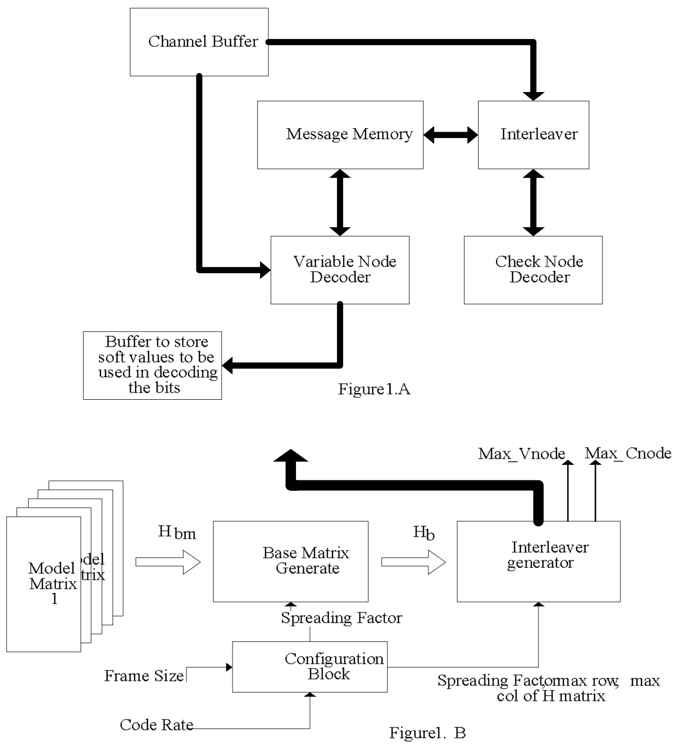

Figure 1: (LDPC Decoder)

$$A_v^i = LLR_v^0 + \sum_{c'\varepsilon C(v)} LLR_{c'v}^i \tag{2}$$

Check Node Processors: Each check node gets the LLR values from the variable nodes to which it is connected and performs parity checks. The valid codeword is the word for which all the parity check equations from all the check nodes are satisfied. The message from check node 'c' to a variable node 'v' is given in equation (3).

$$LLR_{cv}^i = \prod_{v'\varepsilon V(c)\backslash v} sign(Z_{v'c}^{i-1}).\phi\left(\sum_{i'\varepsilon V_j\backslash i} \phi(Z_{v'c}^{i-1})\right)$$

$$\phi(x) = -\log[\tanh(x/2)] = \log\left(\frac{e^x+1}{e^x-1}\right) \tag{3}$$

(The V(c) represents the set of neighboring variable nodes of a check node c)

There is an approximation for (3) similar to the Max-Log-MAP algorithm known as min-sum decoder

$$LLR_{cv}^{i} \cong \min_{v'\varepsilon V(c)\backslash v} \left|Z_{v'c}^{i-1}\right| \cdot \prod_{v'\varepsilon V(c)\backslash v} sign(Z_{v'c}^{i-1}) \qquad (4)$$

The performance of the min-sum is improved with density evolution of *β=0.25*

$$LLR_{cv}^{i} \leftarrow sign(LLR_{cv}^{i}) \cdot \max(\left|LLR_{cv}^{i}\right| - \beta, 0) \quad (5)$$

A. Real Time Programmable LDPC Decoder

A real time Programmable LDPC Decoder is shown in Figure 1. It has two sections. The first section shown in Figure 1.B generates the interleaver and the maximum count for the check and variable nodes while the second section shown in Figure 1.A performs the actual decoding based on the information got from section 1.B. First Base Matrix is generated from the model matrices depending upon a particular code length and code rate. The base matrix is then applied to the interleaver generation block to generate interleaver as well as the maximum count for the check and variable nodes. They are then applied to concerned blocks in section 1.A for real time configuration.

II. Simulation of Algorithms for Decoding IEEE Defined LDPC Codes

The encoder and decoder are simulated for sample code length of 576, rate ½. The Eb/No values chosen is from 0 to 3.5 with 10,000 frames used for a particular Eb/No value. The algorithms simulated include Min-Sum BPA, LLR-BPA, Offset BPA with density evolution factors of 0.15 and 0.25. The performance of decoding algorithms depends upon the structure of LDPC codes. Therefore, the results shown in Figure 2-3 provide performance figures for the irregular code as specified in the IEEE P802.16e/D7 specification. Figure 3 provides BER performance of Min-Sum SPA for varying number of maximum decoder iterations. Four different values were chosen for the maximum decoder iterations and they are 5, 10, 15 and 50. The 10 iterations provide acceptable BER performance and can be selected as the maximum decoder iteration for implementation on either DSP or a reconfigurable fabric. Figure 2 provides results for different decoder algorithms keeping the maximum iteration constant at 10. The algorithm chosen for optimization is the Min-Sum SPA with density evolution of 0.25.

The performance of decoding algorithm varies from one type of LDPC coding to another and is therefore dependent upon the type of coding selected. Since this is the first kind of implementation for this type of coding, exact comparison cannot be carried out due to non-availability of the previous work done on the LDPC coding for WiMax applications. However, approximate comparison can be made with the implementations carried out in the literature. For example, [3] has carried out an FPGA implementation of the Min-Sum SPA algorithm and its modified version. The result of the Min-Sum as depicted in Figure 2 is comparable with the result of the improved min-sum algorithm in [3] though the result in [3] has been obtained for 20 iterations as opposed to the 10 iterations.

Another implementation of the Min-Sum algorithm for

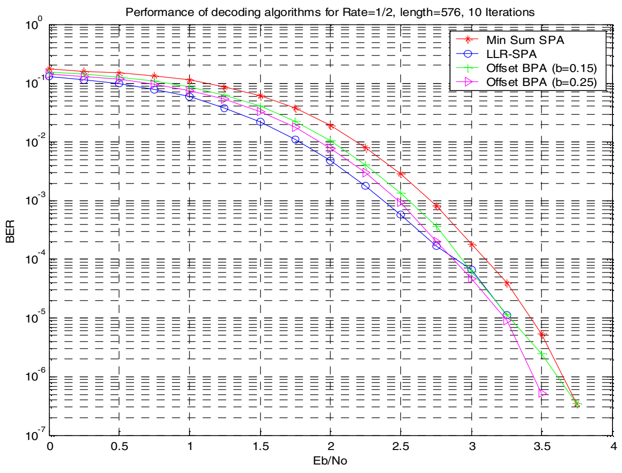

Figure 2: (Performance of different decoding algorithms)

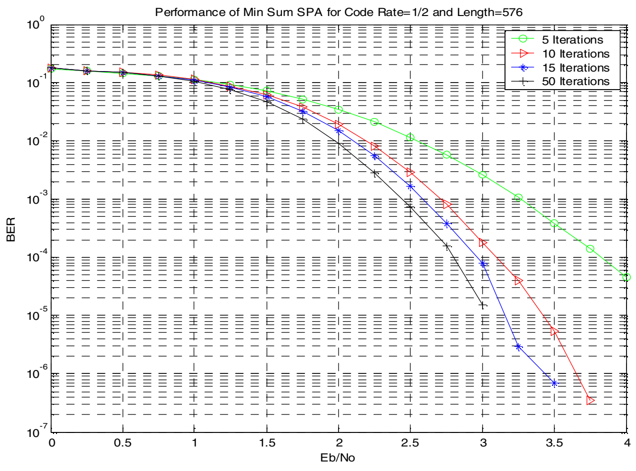

Figure 3: (BER Performance of Min-Sum Algorithm under different decoding iterations)

irregular LDPC code with code length 1268 is done in [4]. The results presented in this report are much better than the result of the ideal min-sum implementation presented in [4]. The same is true with the result shown in [5]. With [6] the results are comparable. The implementation in [7] results in better performance compared to this implementation. This can be the result of density evolution function which is most suited for that particular type of LDPC coding.

Overall there are merits and demerits of this implementation compared to the work in the literature. As far as the ideal Min-Sum is concerned the performance is either better or comparable to that in the literature. However, certain implementations of the modified Min-Sum are better than this implementation but this should not be taken as the argument to question this implementation as it is targeted for different types of LDPC coding.

III. Optimization on The Reconfigurable Architecture (RA)

The simulation results for the un-optimized 'C' code on the RA are presented below:

Time for calculating H matrix and Interleaver	= 14925.8 µsec
Steps taken	= 2141433
Initial execution time for the decoder	= 15062.7 µsec
Steps taken	= 2165765
Execution Time per iteration for the decoding blocks used in actual decoding	= 135 µsec
Throughput per iteration	= 572/135 = 4.3Mbps
Total number of iterations	= 10
Net throughput	= 0.43 Mbps
Throughput per iteration of StarCore	= 9 Mbps
Net throughput on StarCore (10 iterations)	= 0.9 Mbps

The straightforward implementation of decoder on the RA proved to be 52% slower than StarCore. The throughput is improved through optimizing check and variable node decoders and using packed computation.

IV. Optimization of variable node decoder

Optimization is carried out using the following techniques.

Loop distribution: The initial implementation of variable node decoder contains only one major loop and the code inside the loop supports all code rates and code lengths. Different number of memory read and write operations are associated with each code rate as well as with different variable nodes which are separated using 'if' statements. This type of coding style though very compact is highly complex for the architecture compiler to optimize due to a large number of conditional 'if' statements and read/ write memory operations. Separate code is written to support a specific code rate. Inside the 'C' code that supports only one code rate, the major loop has been broken into several sub-loops. For the case of ½ code rate, the major loop that iterates a maximum of 576 times, has been broken into three sub-loops: These support processing of 4, 7 and 3 messages. Loop distribution has facilitated further optimization and reduced some redundant statements and branching.

Hardware Multiplexing: The use of 'if' statement causes jumps which not only time but also power consuming in the cell based architecture. The 'if' conditional statements are reduced to as much as possible and the irreducible ones are replaced with multiplexers.

Memory Access Reduction: In variable node decoder, four different memory arrays are used. They are PtrtoexplicitVnDist, Ptrtoy, Message_Memory and PtrtoLQi. The first stores the number of check nodes that are connected to a particular variable node. The second stores the channel symbols. The third stores LLR messages from variable to check nodes and vice versa. The fourth stores the LLR for the code bits. Out of the four, two arrays PtrtoexplicitVnDist, Ptrtoy are always read, PtrtoLQi is always written while Message_Memory is read as well as written in any iteration. The array PtrtoexplicitVnDist can be removed by using the Loop distribution. Since separate sub-loops are used. These sub-loops have the priori information about the number of messages to be read or written into the Message_Memory, hence the array can be avoided. The removal causes significant reduction in execution time.

Parallel execution of Variable node processors: The architecture has 16 memory read and write interfaces that can support 16 bytes of data to be read as well as written in one memory access time. This allows us to implement several variable node processors in parallel by unrolling the sub-loops. E.g. we can unroll the sub-loop that reads two messages from the Message_Memory as well as one from the Ptrtoy array by 5 and can read the 10 bytes from Message_Memory in one memory access time and the 5 bytes from the Ptrtoy array in another memory access time. Thus, 5 two-message variable node processor (Figure 3) can be executed in parallel. The number of adders/subtractors, shift registers and temporary registers are also increased to accommodate the parallel execution of the variable nodes. This caused tremendous reduction in execution time.

With these optimizations, the execution time for processing 576 variable node processors (frame size is 576) came out to be 12.124 µsec per iteration.

Pipelining the code: Each variable node processor has three operations: memory read, computation and memory write. If they are pipelined, the architecture can be clocked at a frequency higher than the frequency used by the combinational counterpart. This will reduce the execution time by almost 3 times. All it needs is to bring the loop inside one step. Within one step, the loop jumps to itself and can be pipelined. The number of resources is increased to bring the loop inside one step. Another necessary condition observed with the RA compiler for bringing the code to one step is to make the limit of the loop a constant. This is done by writing separate functions for each code length. This increased the size of the code but also made it possible to bring the code inside one step which is necessary for throughput enhancement.

The step is pipelined by inserting registers between memory read, computation and memory wirte blocks. The pipelined 'C' code runs three times faster. With pipelining, a reduction of 2.5 times has been achieved. The execution time for the VNode_Decoder came out to be 4.85 µsec per iteration. The Check Node Decoder is similarly optimized.

Pipelining is also used in the initial memory setup and code write up. The overall execution time is is given below

C Node Decoder	= 3.5	µsec per iteration
V Node Decoder	= 4.85	µsec per iteration
Interleaver	= 0.72	µsec
Initial Memory Setup	= 4.84	µsec
Code Write Up	= 1.28	µsec

The overall execution time for 10 iterations is (3.5+4.85+0.72)*10+4.84+1.28=96.82µsec.This corresponds to a 5.94Mbps. This is now 6.6 times the speed achieved with SC140 and equivalent to the speed achieved in [7].

V. The idea of packed computation

The RA is using bus width of 32 bits for each of its cell. In ASIC implementation of LDPC decoding the data width normally used is 6 bits with 1-bit for sign, 2 bits for whole number and 3 bits for precision. In DSP applications, 8 bits can be used instead. This implies that with a 32-bit cell only the first 8 bits will be doing useful operation while the remaining 24 bits will be idle. If the configuration is chosen such that the 32-bit cell can act as a single 32-bit cell or two independent 16-bit cells or four independent 8-bit cells, then a single cell can be used for up to four same operations which would otherwise require the use of four 32-bit cells.

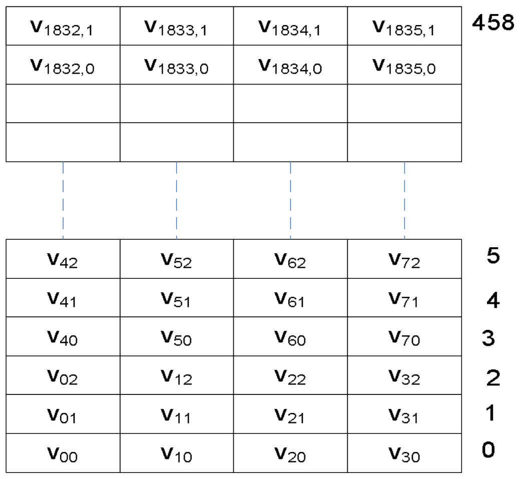

Figure 4: (Variable Node Message Distribution)

The packed computation will be used to execute four variable node processors in parallel. The idea is to pack the four 8-bit soft values from the channel in one integer and store the integer in four consecutive memory banks through a one 32-bit memory interface. When the integer value is read, it will have soft values for the four consecutive variable nodes. These integer values will automatically get divided among four 8-bit values and will be processed in the logic independently. The distribution of the messages inside the four banked Message Memory for the variable node decoder is given in Figure 4. Here $v_{i,j}$ represents the messages from variable to check nodes with 'i' being the number of the variable node and 'j' the number of the messages required to read or write by the i-th variable node. With this arrangement, it is possible to read 4 messages for 4 variable node decoders in just one memory read cycle. If we use 8 memory read/write interfaces, then the code inside each sub-loop can be brought inside one step for possible pipelining. The speed of the variable node processing can be increased four times due to parallel execution of four nodes. This implies that variable node processing can be completed in 4.85/4 = 1.2 μsec per iteration.

After processing the variable node, interleaver is used to rearrange the messages inside the Message Memory. After interleaving the messages, the Message Memory looks like as shown in Figure 5. Here $c_{k,m}$ represents the messages to check node decoder with k being the number of the check node decoder and m being the messages that the kth check node decoder are reading from or writing to the Message Memory. Since the check node decoder is divided into two sub-loops: one processes six while the other processes seven messages at a time. The execution time previously calculated is 3.5 μsec. With packed computation, it can be reduced to 3.5/4 = 0.875 μsec per iteration. The memory initialization and code write up would ideally take 4.84/4=1.21 μsec and 1.28/4=0.32 μsec. The new execution time would be

C Node Decoder	= 0.875	μsec per iteration
V Node Decoder	= 1.21	μsec per iteration
Interleaver	= 0.72	μsec per iteration
Initial Memory Setup	= 1.21	μsec
Code Write Up	= 0.32	μsec

The overall execution time will be 29.58 μsec. This is equivalent to 20 Mbps.

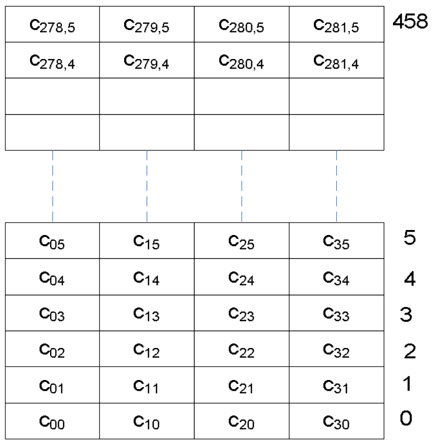

Figure 5: (Check Node Message Distribution)

VI. CONCLUSION

A real time LDPC decoder for Mobile WiMax applications as stated in IEEE P802.16E standard has been implemented and optimized on the reconfigurable instruction cell architecture. Several general purpose and architecture specific optimization techniques have been applied for throughput improvement. We have been able to achieve 20Mbps throughput with these optimizations. This is preliminary work on optimizing 'C' code on the architecture. The architecture can provide a throughput of as much as 100Mbps subject to putting enough resources on the reconfigurable fabric.

VII. REFERENCES

[1] IEEE P802.16E/D7 Specification published in 2006

[2] Ying Yi; Nousias, I.; Milward, M.; Khawam, S.; Arslan, T.; Lindsay, I."System-level Scheduling on Instruction Cell Based Reconfigurable Systems", Design, Automation and Test in Europe, 2006. DATE '06. Proceedings, Volume 1, 6-10 March 2006 Page(s):1 - 6

[3] Shimizu, K., Ishikawa, T., Ikenaga, T., Goto, S., Togawa, N.,"Partially-Parallel LDPC Decoder Based on High-Efficiency Message-Passing Algorithm", Computer Design, 2005. Proceedings. 2005 International Conference on 02-05 Oct. 2005 Page(s):503 - 510

[4] Zarkeshvari, F., Banihashemi, A.H.,"On implementation of min-sum algorithm for decoding low-density parity-check (LDPC) codes", Global Telecommunications Conference, 2002. GLOBECOM '02. IEEE Volume 2, 17-21 Nov. 2002 Page(s):1349 - 1353 vol.2

[5] Anastasopoulos, A., "A comparison between the sum-product and the min-sum iterative detection algorithms based on density evolution", Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE Volume 2, 25-29 Nov. 2001 Page(s):1021 - 1025 vol.2

[6] Ryan-crc-ldpc-chap.pdf

[7] Lechner, G.; Sayir, J.; Rupp, M.;,"Efficient DSP implementation of an LDPC decoder", Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on Volume 4, 17-21 May 2004 Page(s):iv-665 - iv-668 vol.4