Quantitative Evaluation of Low Density Parity Check Convolutional Code Encoder and Decoder Algorithms for the XInC MIMD Multithreaded Microprocessor

Xin Sheng Zhou, Bruce Cockburn, Stephen Bates
Department of Electrical and Computer Engineering
University of Alberta
Edmonton, AB, Canada T6G 2V4
xzhou, cockburn, sbates@ece.ualberta.ca

Abstract—In this paper, we introduce Low Density Parity Check Convolutional Code (LDPC-CC) parallel encoder and decoder algorithms for the XInC MIMD multi-threaded microprocessor. A modified memory-based decoder architecture and an interleaved LDPC-CC code scheme are also proposed. Extensions and simple modifications to the current XInC microprocessor are proposed to decrease the number of instruction cycles per decoded bit. A XInC emulator was built to evaluate and quantify the hardware utilization and performance benefits of these modifications and other alternatives.

Keywords: convolutional codes; low density parity check; Multiple Instruction Multiple Data architecture

I. INTRODUCTION

Low density parity check block codes (LDPC-BCs) were first proposed by Gallager in the 1960’s [1]. However, the construction that he proposed was not very elegant, and the performance that he reported was not that impressive and therefore his contribution was largely ignored for almost thirty years. In 1996, LDPC-BCs were rediscovered by Mackay and Neal and shown to have error correction performance approaching the Shannon capacity limit [2]. In 1999, Felström and Zigangarov proposed low density parity check convolutional codes (LDPC-CCs). It has been shown that LDPC-CCs have better code performance than the regular LDPC-BCs with the same memory length constraints. Furthermore, LDPC-CCs have a simpler encoder algorithm and the decoder can be implemented by multiple identical processors in a pipeline structure [3]. Moreover, LDPC-CCs allow data sequences of arbitrary length to be encoded, making them much more convenient for efficient packing into the payload field of the available data frames as well as for streaming data applications [4].

The decoder algorithm complexity mainly comes from the soft-in-soft-out decoding processing and the need for multiple iterations for decoding. VLSI technology makes possible parallel decoding algorithms and approximate decoding of high-throughput LDPC codes with good code performance. Currently, microprocessor implementations of LDPC-BCs can achieve a throughput of 100K bps while custom ASIC implementation can achieve up to 1G bps [6].

Though the current throughput of microprocessor implementations is not very impressive, the benefits of these implementations are faster time-to-market deployment and higher flexibility. To increase the throughput of the decoding algorithm by exploiting the available parallelism, parallel decoding algorithm could be used. Quantitative evaluation of the decoding algorithm could also help to determine the bottleneck in the algorithms, so microprocessor hardware resource could be added or modified to eliminate the bottleneck.

II. LOW DENSITY PARITY CHECK CONVOLUTIONAL CODES

LDPC-CCs generate code bits based on a low density parity check matrix. Unlike LDPC-BCs, which generate code bits based on the information bits only, LDPC-CCs are generated from both previous information bits and previous code bits. In this paper, we construct a (128,3,6) LDPC-CC. The most recent 128 information bits and 128 code bits are stored in a First-in First-out (FIFO) queue. Each parity check operation involves 6 bits. Each bit involves 3 parity check operations. This rate 1/2 LDPC-CC can be constructed as follows,

\[ v_1(t) = u(t) \]
\[ v_2(t) = \sum_{i=0}^{M} h_1^{(i)}(t)u(t-i) + \sum_{i=1}^{M} h_2^{(i)}(t)v_2(t-i) \]

where \( u(t) \) is the information bit sequence, \( v(t) \) is the code bit sequence. When the encoder reads one information bit, it generates two code bits accordingly. The first code bit is the parity check result generated from both previous information bits and previous code bits in the memory queue. The parity check operation is a simple XOR operation. Vector \( h_1(t) \) specifies which bits in the queue are involved in the same parity check operation. The positions of those operation bits in
the queue are different for each parity check operation, but this pattern repeats periodically. Therefore, a position table is also maintained.

LDPC-CCs could also be defined by their infinite, but periodic parity check matrix $H$. The transpose $H^T$ can be represented as

$$
H^T = \begin{bmatrix}
h_1(0) & h_1(1) & \cdots & h_1(M) & 0 & \cdots \\
h_2(0) & h_2(1) & \cdots & h_2(M) & 0 & \cdots \\
\vdots & \vdots & \ddots & \vdots & \vdots & \ddots \\
0 & 0 & \cdots & h_2(M) & 0 & \cdots \\
0 & 0 & \cdots & 0 & 0 & \cdots \\
\end{bmatrix}
$$

(3)

For a rate 1/2 code, an information sequence $u = [u(0), u(1), \cdots, u(t)]$

is encoded into a sequence $v = [v_1(0), v_2(0), v_1(1), v_2(2), \cdots, v_1(t), v_2(t)]$

such that

$$v \bullet H^T = 0$$

(6)

The parity structure of an LDPC-CC can be represented graphically by a Tanner graph [8] as shown in Fig. 1. There is one variable node for each bit and there is one check node for each of the parity check operations. The edges between each check node and the variable nodes specify which bits are involved in the same parity check operation.

B. Decoder Algorithm

The architecture of an LDPC-CC decoder has a simple unidirectional cascade of processors. Each processor is comparable to an iteration operation. In fact, performance increases as we add more processors to the cascade. Fig. 3 shows the LDPC-CC decoder architecture.

Inside the processors, a memory block is used to store the results of the check node and variable node operations. The first row of the memory stores the input channel Log Likelihood Ratio (LLR) values. Other rows store the values from the check node operation results. In our regular (128,3,6) code, each bit involves three parity check operations, therefore, there are three additional rows to store the check node results. At the end of the processors, a variable node operation performs to sum up the check node results.

The inputs of the first processor are information bit LLRs and code bit LLRs. The inputs of the following processors are the outputs of the previous processors. At the last processor, a hard decision operation on the LLR sign produces the output digit ‘0’ or ‘1’.

Log Likelihood Ratios

$$l(x) = \log \frac{P[t(k) = '0'] r(k) = x]}{P[t(k) = '1'] r(k) = x]}$$

(7)

are used for the probabilities of the bit value. $t(k)$ denotes the transmitted bits. The values of the received bits $r(k)$ are $x$. The advantage of using LLR value is that the product operation of the probability can be replaced by simpler addition operation. For an Additive White Gaussian Noise (AWGN) Binary Phase-shift Keying (BPSK) channel, the LLRs would be...
where \( \sigma \) is the standard deviation of the AWGN.

The min-sum algorithm [9] is used to approximate the statistically exact complicated \( \tanh/\tanh^{-1} \) functions in check node operation.

For the check node,\n\[
y_k = \min_{n \neq k} \{ |x_n| \} \prod \text{sgn}(x_n), \tag{9}
\]
where\n\[
n, k \in \{1,2,...6\}.
\]

Min(.) is the absolute minimum value of all the other LLRs received by check node. Sgn(.) is the sign function. The value is ‘1’ when \( x_n \) is a positive number and is ‘-1’ when \( x_n \) is a negative number.

For variable node, \( y_k \) is the sum of all the other LLRs received by variable node,
\[
y_k = \sum_{n \neq k} x_n, \tag{10}
\]
where\n\[
k \in \{2,3,4\}, n \in \{1,2,3,4\}.
\]

IV. ARCHITECTURE OF XINC MIMD MICROPROCESSOR

XInC is a 16-bit pipelined RISC microprocessor with 8 hardware thread processors and a shared memory space. Each thread processor has its own 8 general purpose registers, 4 condition code registers and program counter. A single 16-bit instruction is executed in a shared 8-stage pipeline. The memory crossbar is responsible for routing the data to the addressed memory blocks. The thread processors have two ports that are connected to the memory crossbar. The Instruction (I) port is used to fetch instructions. The data (D) port is used to access data. A hardware semaphore mechanism is used to manage the shared resources such as memory and I/O ports [5]. Fig.4 shows the architecture of XInC microprocessor and its memory model.

V. PARALLEL ALGORITHM ARCHITECTURE

Fig. 5 shows our LDPC-CC encoder and decoder architecture for the XInC microprocessor. One of the thread processors is used for the encoder. Another thread processor is used for the decoder controller, data input/output and hard decision operation. The remaining six thread processors are used for six cascade LDPC-CC decoder processors, which each includes one check node operation and one variable node operation.

In this memory-based decoder architecture, each LDPC-CC processor accesses its own memory block. Therefore, there are no memory access conflicts among the LDPC-CC decoder processors and no semaphore mechanism is needed to manage the shared memory resources.

Figure 4. XInC microprocessor architecture

Figure 5. LDPC-CC parallel algorithm and memory block architecture

In thread processor 2, decoder controller performs data input/output, output data hard decision and thread control. In our design, we allocate additional memory in the memory queue to store the input and output data. Because of this separated memory, the data input/output and hard decision operations can be performed when the LDPC-CC decoder processors are running without memory access conflicts. The instruction cycles per decoded bit are decreased by 40% compared with the previous memory-based architecture [7]. Fig. 5 shows the decoder memory architecture.
To further increase the decoder throughput, interleaved LDPC-CCs could be used. At the encoder end, the information bits are interleaved to several streams. Each stream can be encoded individually and combined before the transmission. At the decoder end, the code bits can be interleaved again and decoded accordingly. Using this approach, the LDPC-CC decoder throughput can be directly scaled up with the available MIMD multithread processor resources. Fig.6 and Fig. 7 shows the interleaved LDPC-CC encoder and decoder data flow.

Circular addressing register is a register whose value has an upper boundary. When this boundary is reached by addition operation, the register value would reset to zero.

<table>
<thead>
<tr>
<th>Operation type</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Looping</td>
<td>38%</td>
</tr>
<tr>
<td>Data moving</td>
<td>19%</td>
</tr>
<tr>
<td>Decoding operation</td>
<td>43%</td>
</tr>
</tbody>
</table>

![Table I. LDPC-CC Decoder Operation Frequency](image)

Next, 19% of the instruction cycles are for data moving between the memory and registers. This large memory-access comes from the check node and variable node operations, which require fetching several data from the memory, calculating and storing the results back to the memory. For variable node operations, the data addresses are continuous. If double word memory access is used, 4 data values could be accessed in one instruction, and then the memory access overhead could be reduced by 75%. For check node operations, the data addresses are separated. If two-port memory read access could be used, the memory access overhead could be reduced by 50%.

Finally, some frequently used decoding operations in the decoder algorithm are saturation operations and minimum number operation. These operations are currently performed by several XInC instructions. If we could design new instructions and hardware for these operations, the decoder algorithm performance could also be increased. However, these new instructions might be application-specific and not be commonly used by other applications.

XInC emulator is designed by object oriented language. Hardware component such as zero overhead looping registers and circular addressing registers could be added to the emulator easily. Also, instructions such as saturation operations and minimum number operations could also be added. The decoder algorithm performance could be evaluated under these new modifications by the XInC emulator.

**VI. XInC Emulator and Quantitative Algorithm Evaluation**

To evaluate the LDPC-CC decoder algorithm on the XInC microprocessor, we designed a XInC emulator. The emulator can execute the instructions in each clock cycles. By using the emulator, we can directly count the decoder instruction cycles per bit. This is the main measurement of the decoder algorithm performance.

The LDPC-CC decoder operation frequency is shown in Table I.

First, as we see here, 38% of the instruction cycles are not used for decoding operation, but for the looping overhead. The loop instructions control how many LLRs should be read from memory for the check nodes and variable nodes. If zero-overhead looping and circular addressing technology commonly used in DSP field were to be implemented on the XInC, the decoder algorithm could run faster.

Zero overhead looping registers are the registers to store the looping counter. For each loop, the register value would reduce 1. When the register value reaches 0, the loop is done. The assembly language code could be,

```
mov zolr0, 6
Loop:
Any instructions
zol zolr0, Loop
```

where zol is the zero overhead looping instruction name, zolr0 specifies the name of the zero overhead looping register, Loop specifies the looping address.

**EVALUATION**

Currently, our LDPC-CC decoder algorithm requires 764 instruction cycles per decoded bit. On a 12-MHz XInC microprocessor, we designed a XInC emulator. The emulator could be used to evaluate the algorithm performance. We can directly count the instruction cycles per decoded bit. On a 12-MHz XInC microprocessor, we designed a XInC emulator. The emulator could be used to evaluate the algorithm performance. We can directly count the instruction cycles per decoded bit. On a 12-MHz XInC microprocessor, we designed a XInC emulator. The emulator could be used to evaluate the algorithm performance. We can directly count the instruction cycles per decoded bit. On a 12-MHz XInC
microprocessor the current decoder throughput is near 2 Kbps. Figure 8 shows the (128,3,6) LDPC-CC code performance on six decoder processors.

VIII. CONCLUSION

In this work, we implemented and evaluated the LDPC-CC encoder and decoder algorithm on the XInC MIMD microprocessor. A parallel decoder algorithm was used to maximize the decoder throughput. The algorithm was quantitative evaluated using the XInC emulator tool. We are currently implementing some new hardware resources and instructions to the XInC emulator and evaluating the decoder performance based on these modifications.

REFERENCES


