Master Thesis

HW/SW Co-design and Implementation of a Fountain Code for an FPGA System-on-Chip

Khurram Ashraf
ashraf@rhrk.uni-kl.de

Supervisors: Prof. Dr. Klaus Schneider
Prof. Dr.-Ing. Norbert Wehn
M. tech. Tripti Jain
Dr. -Ing. Christian Weis
Declaration

This work is the result of my own effort, and has been completed without any external help other than mentioned in this project report. References and literature that I have used are fully listed without emission in the literature list of my project.

Khurram Ashraf

TU Kaiserslautern

29 September, 2016.
Acknowledgment

In doing this thesis work, I was blessed to work with some truly remarkable people. First of all, I would like to express my sincere gratitude to Miss Tripti Jain, Dr.-Ing. Christian Weis and Dipl.-Ing. Stefan Scholl for their supervision and support during the entire period of this thesis.

I am really thankful to Prof. Dr. Klaus Schneider and Prof. Dr.-Ing. Norbert Wehn for their support by providing me an opportunity to work in their research group of Embedded Systems and Microelectronic Systems Design at Technical University of Kaiserslatuern. Furthermore, the knowledge I acquired by taking courses under their supervision helped me in building in-depth understanding for this thesis.

Lastly, but most importantly, I would like to express my gratitude to my parents and brother for helping me during my studies.
Abstract

Wireless communication traffic has increased significantly over the past few decades. Communication via noisy and unreliable channel can lead to data loss or erroneous data information. This problem can be resolved using a back channel that allows a receiver to send a re-transmission request. However, in certain scenarios, i.e. deep space communication and point-to-multipoint communication, re-transmission can be expensive and leads to significant communication overhead. To overcome this problem Forward Error Correction (FEC), also called channel code, are used.

FEC allows the sender to encode the message in a redundant fashion by using Error Correction Code (ECC). This redundancy allows the receiver to correct the errors that occurred during transmission without asking for re-transmission.

The thesis presents an analysis of Fountain code; Fountain code is a class of FEC. In designing embedded systems, parameters such as timing performance, energy consumption, throughput, and latency are of great importance. In this thesis work different types of Fountain codes are implemented on an embedded platform (Zynq platform) to find a balance between hardware and software in terms of timing, energy, throughput and latency. Initially, Fountain codes are implemented on an embedded processor (ARM cortex A9), and later on, efficient partitioning is sought between the hardware and software. Finally, an analysis is carried out for all the implementations in terms of timing, energy, throughput, and latency.
# Contents

## 1 Introduction
1.1 Motivation .................................................. 1
1.2 Zed Evaluation Board ........................................ 4
1.3 Main Goals and Tasks ...................................... 5
1.4 Related Work ............................................... 7
1.5 Organization of the Thesis .................................. 7

## 2 Fundamentals of Fountain Code
2.1 Channel ...................................................... 9
  2.1.1 Binary Erasure Channel .................................. 9
  2.1.2 Binary Symmetric Channel .............................. 10
2.2 Forward Error Correction .................................. 11
2.3 Linear Block Codes ........................................ 11
2.4 Systematic and non-Systematic Codes ..................... 11
  2.4.1 Systematic Codes ...................................... 12
  2.4.2 Non Systematic code .................................. 14
2.5 Low Density Parity Check Codes(LDPC) ................. 14
2.6 Message Passing Algorithm ................................. 14
2.7 LT-Code .................................................. 18
  2.7.1 LT-Encoder ........................................... 19
  2.7.2 LT-Decoder .......................................... 20
2.8 Raptor Code ............................................... 22
  2.8.1 Raptor Encoder ...................................... 24
  2.8.2 Raptor Decoder ....................................... 26

## 3 HW-SW Co-Design of Fountain Code on Zynq platform
3.1 LT-Code Encoding on Embedded Processor ............... 29
3.2 LT-Code Decoder on Embedded Processor ................. 31
# List of Figures

1.1 Communication system of LT-code. ..................................... 2  
1.2 Communication system of Raptor code. ................................. 3  
1.3 Top view of Zedboard [1]. .............................................. 5  
1.4 Block Diagram of Zedboard [1]. ....................................... 6  

2.1 Binary Erasure Channel .................................................. 10  
2.2 Binary Systematic Channel .............................................. 10  
2.3 Systematic Code .......................................................... 12  
2.4 Bipatriate graph of LDPC code ......................................... 15  
2.5 Encoding to produce check nodes. ..................................... 16  
2.6 Decoding using the message passing algorithm for the received string $y = [10xxx1]$. Sub-figures show the decision taken in each step. ................................................................. 17  
2.7 Generator Matrix. Purple column shows the packets erased by the BEC. ................................................................. 19  
2.8 Generator Matrix at the receiver side. Where $N \geq K$ .............. 20  
2.9 LT-Enoder ................................................................. 21  
2.10 Packet received at LT-Decoder. Rest of packets are deleted by the channel. ............................................................ 22  
2.11 Example step by step decoding of LT-Code. Here $K = 4$ and $N=6$. ................................................................. 23  
2.12 Raptor code ................................................................. 24  
2.13 Raptor encoding: Intermediate symbols are generated in the first step. Later, they are used by LT-code as an input to generate output. .................................................. 25  
2.14 Raptor Decoding: (a) Intermediate symbols are retrieved, but one symbol could not be retrieved. (b) Missing symbol was retrieved by the outer code. .................................. 27
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Time consumed by an LT-encoder to encode for different data size.</td>
<td>29</td>
</tr>
<tr>
<td>3.2</td>
<td>Flow chart of LT-encoder on embedded processor.</td>
<td>30</td>
</tr>
<tr>
<td>3.3</td>
<td>Energy consumed by an LT-encoder to encode for different data size.</td>
<td>31</td>
</tr>
<tr>
<td>3.4</td>
<td>Flow graph of LT-Decoder.</td>
<td>32</td>
</tr>
<tr>
<td>3.5</td>
<td>LT-code decoding. Where $K$ represents number of packets of original message. Here $T = 4$ bytes</td>
<td>33</td>
</tr>
<tr>
<td>3.6</td>
<td>LT-code decoding. Here $T = 64$ bytes</td>
<td>33</td>
</tr>
<tr>
<td>3.7</td>
<td>Energy Consumption of LT-code decoding. Here $T = 4$ bytes</td>
<td>34</td>
</tr>
<tr>
<td>3.8</td>
<td>Energy Consumption of LT-code decoding. Here $T = 64$ bytes</td>
<td>35</td>
</tr>
<tr>
<td>3.9</td>
<td>Flow chart of Raptor-encoder.</td>
<td>36</td>
</tr>
<tr>
<td>3.10</td>
<td>Time consumed to generate intermediate packets by Raptor encoder.</td>
<td>38</td>
</tr>
<tr>
<td>3.11</td>
<td>Time consumed to generate output packets by Raptor encoder.</td>
<td>38</td>
</tr>
<tr>
<td>3.12</td>
<td>Raptor code decoding time consumption on embedded platform.</td>
<td>40</td>
</tr>
<tr>
<td>3.13</td>
<td>Raptor code decoding energy consumption on embedded platform.</td>
<td>40</td>
</tr>
<tr>
<td>3.14</td>
<td>Flow chart of Raptor decoder.</td>
<td>41</td>
</tr>
<tr>
<td>3.15</td>
<td>Decoding time of LT-decoder using hardware accelerator.</td>
<td>42</td>
</tr>
<tr>
<td>3.16</td>
<td>Energy consumed by the hardware accelerator.</td>
<td>43</td>
</tr>
<tr>
<td>3.17</td>
<td>Resourced utilized by the hardware.</td>
<td>43</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Extra packets generated by the encoder of Raptor code. . . . . 37
Chapter 1

Introduction

This chapter focuses on the main goals of the thesis. A major emphasis has been donned over Fountain code which is used as a core code throughout the thesis.

1.1 Motivation

Wireless communication traffic has increased rapidly over the past few decades. Communication over unreliable and noisy channel can lead to significant amount of data loss or erroneous data. To ensure that data has been sent correctly, a back channel is required which is used by the receiver for re-transmission requests. Re-transmission leads to significant overhead and is expensive, e.g. deep space communication, mobile communication, etc.

To overcome this problem, different coding techniques are proposed. These coding techniques are called channel coding or Forward Error Correction (FEC). FEC codes are a solution where re-transmission delays cannot be afforded by a communication system. They ensure that the receiver itself corrects the data transmission errors, hence ceasing the need for a re-transmission.

Fountain code - a class of FEC - is implemented in this thesis. Normally, codes using FEC exhibit some fixed rate. The rate is a ratio between encoded symbols and input symbols. However, Fountain codes do not have fixed rate. They are often called rateless erasure codes due to this reason.
1.1. MOTIVATION

As the name suggests, Fountain codes are analogous to water droplet in a fountain. They produce an infinite number of code-words from a given input data, where each code-word can be seen as a droplet. At the decoder side message $M$, which is divided into $K$ numbers of symbols (each symbol $T$ is of $n$ bits) can be decoded from $N$ symbols. Where $N$ is any subset of symbols which are produced by the encoder at the receiver side. However, $N$ should be greater than or equal to $K$.

Fountain codes have two important classes of codes, namely LT-code and Raptor code.

LT-code allows the encoder to produce potentially infinite stream of encoded symbols, hence the decoder can decode the original message from any subset of the encoded message it receives. Figure 1.1 shows a communication system of LT-code.
system of LT-code. However, LT-code do not have linear time encoding and decoding complexity. To overcome this problem another type of Fountain code was introduced called Raptor code.

Raptor code performs encoding and decoding in linear time. It introduces a new stage in LT-code called outercode/pre-code. At the receiver side, the message is initially encoded by an outercode that produces intermediate symbols. Later on, these intermediate symbols are encoded using weakened LT-code into output symbols. At the decoder side, symbols that are not recoverable by weakened LT-code are recovered by the outercode. Figure 1.2 shows a communication system of Raptor code.

Figure 1.2: Communication system of Raptor code.
1.2. ZED EVALUATION BOARD

In designing an embedded system, parameters such as timing performance, energy consumption, throughput and latency are of great importance. The main objective of this thesis is to explore the design space of Fountain code for an embedded system (Zynq Platform) and to find a balance between hardware and software. Initially, Fountain codes are implemented as software on an embedded platform to realize different parameters for the implementations. Based on these factors, an optimum partitioning between hardware and software is sought to attain maximum benefit in terms of timing, energy consumption latency and throughput (as in most cases high throughput and low latency is required with low energy consumption). Furthermore, the aim also includes analysis of overall gain using different implementations (software only and one with hardware accelerator).

1.2 Zed Evaluation Board

Zed board is a prototyping and evaluation kit consisting of Zynq 7000, All Programmable System-on-Chip (AP SoC) as shown in Figure 1.3. It contains various peripheral devices that makes it suitable for various applications. It can be used efficiently in scenarios where hardware software co-designing (HW/SW co-design) is required, whilst allowing intensive processing capability and higher energy efficiency. Figure 1.4 shows the block diagram of Zedboard.

Zynq 7000 is divided into two different parts, namely Processing System (PS) and Programmable Logic (PL). It contains a Configuration Logic (CL) interface with ARM dual core Cortex A9 processing system [2].

The tools used in this thesis to work on Zedboard are Vivado and Vivado HLS. Vivado and Vivado HLS are software suits from Xilinx used for developing System-on-Chip (SoC) and also for High Level Synthesis (HLS).
1.3. MAIN GOALS AND TASKS

In this thesis work we will focus on resources (such as time consumption, power consumption, latency, and throughput) which are utilized for encoding and decoding different types of Fountain codes by doing hardware software co-design. Two different Fountain codes will be analyzed in this work, namely LT-code and Raptor code. Following are the main tasks implemented in this thesis:

- Implementation of LT code for an embedded processor on FPGA (Zynq platform).
- Checking for latency, throughput, and energy consumption for the implementation.
- Deciding which part of the software can be implemented in hardware to accelerate.
- Implementation of hardware in FPGA.
1.3. MAIN GOALS AND TASKS

- Comparing different implementations to find a balance between hardware and software (software implementation of LT code with the implementation having hardware accelerator).

- Implementation of Raptor code on an embedded processor.

- Check for latency, throughput and energy consumption for Raptor code implementation.

- Deciding which part of the software can be implemented in hardware to accelerate.

- Implementation of hardware on FPGA.

- Comparing different implementations to find a balance between hardware and software.

Figure 1.4: Block Diagram of Zedboard [1].
1.4 Related Work

Todor Mladenov et al [3][4] were the first to realize resources (timing, power consumption, etc.) used for encoding and decoding when Fountain codes are implemented on embedded platform. Furthermore, they also realized the results when certain part of software is used along with hardware accelerator on embedded platform (accelerating certain parts of the software which consumes most of the time).

Todor Mladenov et al [3][4] used NIOS soft-core processor on Altera Stratix FPGA. Initially, Raptor code was implemented in software with different packets sizes K and each packet T containing \(x\) bits. Later on, the algorithm was implemented in hardware. It was noted that the hardware achieves 5.9 times better performance in terms of timing and 5.5 times in terms of energy consumption for low values of T. However, for larger values of T, the timing performance of hardware is 6.9 times that of the software.

Hence, it was shown that using a hardware with a lower value of K and a higher value of T can lead to better performance. However, a significant improvement cannot be achieved by the hardware.

1.5 Organization of the Thesis

This thesis work is organized as follows. In Chapter 2, fundamentals of Fountain code are explained. Two different types of Fountain codes namely, LT-code and Raptor codes, are explained in detail. Step by step encoding and decoding process of LT-code and Raptor code is presented. Chapter 3, presents the hardware software co-design of LT-Code and Raptor code. Results of software implementation on embedded processor (ARM Cortex A-9) are presented for the two algorithms. Furthermore, hardware implementation results are also presented for certain part of the software to analyze the difference between both the implementations. In chapter 4, results of the
1.5. ORGANIZATION OF THE THESIS

implementations are discussed in detail. Finally, Chapter 5 concludes the report by presenting a brief conclusion of the work.
Chapter 2

Fundamentals of Fountain Code

This chapter presents the fundamentals of Fountain code. It introduces two types of Fountain codes, LT-code and Raptor code and explains the encoding and decoding process of both the codes.

2.1 Channel

A *channel* basically refers to a transmission medium between sender and receiver. Channel models simplify the understanding of communication between sender and receiver. In the context of Fountain codes, Binary Erasure Channel (BEC) and Binary Systematic Channel (BSC) are the most important channel models.

2.1.1 Binary Erasure Channel

Binary Erasure Channel (BEC) is a simple model for analysis, yet it is quite useful and used frequently. In this model, the transmitter transmits the data, whilst on the receiving side, received data is either the correct data which was sent or it will be erased.

Suppose a sender sends 0 or 1, the receiver will either receive 0 or 1 respectively, depending upon the data transmitted. However, if the data is erroneous, the data will be erased as shown in Figure 2.1. Here “$\alpha_e$” is called *erasure probability*. Hence, in this channel only “1- $\alpha_e$” of data packets will be recovered.
2.1. CHANNEL

2.1.2 Binary Symmetric Channel

Binary Symmetric Channel is another simple model which is used frequently in coding theory. In this model, there is a small probability that the receiver might receive erroneous message from the transmitter.

Suppose a transmitter sends 0 or 1 and the receiver receives 0 or 1 correctly depending upon the data sent. In case of error, the receiver will receive erroneous values as shown in Figure 2.2. Here, “α_e” is called error probability or crossover probability. Hence, for this channel error free receiving probability will be “1- α_e”.

Figure 2.1: Binary Erasure Channel

Figure 2.2: Binary Systematic Channel
2.2. FORWARD ERROR CORRECTION

The conditional probabilities of the channel are as follows:

\[ P (Y=0 \mid X=0) = 1- \alpha_e \]
\[ P (Y=1 \mid X=0) = \alpha_e \]
\[ P (Y=0 \mid X=1) = \alpha_e \]
\[ P (Y=1 \mid X=1) = 1- \alpha_e \]

2.2 Forward Error Correction

Transmitting data over unreliable communication channel can be very tricky while ensuring correct transmission of the data. For this purpose redundancy is added while transmitting the data, to ensure that the receiver itself can correct the received data if some errors occurred. Hence, a back channel is not required by the receiver for re-transmission request. Methods used for encoding data in a redundant way is called Forward Error Correction (FEC) codes or channel codes.

2.3 Linear Block Codes

Block Codes are error correction codes that encodes an input block of size \( k \) to an output block of size \( n \). Here \( k \) is called message length whereas \( n \) is called block length. The rate of the code is a ratio between message length and block length (\( r = k/n \)).

*Linear block codes* is a type of error correcting codes, where any linear combination of codewords is also a codeword. Often, such type of block codes are expressed in terms of matrix called *generator matrix*.

2.4 Systematic and non-Systematic Codes

Systematic and non-systematic codes tell us about the arrangement of the packets.
2.4. SYSTEMATIC AND NON-SYSTEMATIC CODES

![Systematic Code](image)

Figure 2.3: Systematic Code

2.4.1 Systematic Codes

Error-correcting codes (ECC) are systematic if the original message is the part of the encoded symbol as shown in Figure 2.3. The first $k$ bits are the original message bits while the remaining $n-k$ bits are the redundant bits. Redundancy bits are used to detect the errors in the message and also to correct it.

Here we consider only binary message containing 0 or 1 [8]. Suppose, the message to be transmitted is $x = [c_0 \ c_1 \ c_2]$. We add some redundancy bits $m = [c_3 \ c_4 \ c_5]$ which are called check bits to generate a codeword of the message. Check bits are added such that they satisfy parity check equation. code C contains $[c_0 \ c_1 \ c_2 \ c_3 \ c_4 \ c_5]$.

\[
\begin{align*}
  c_0 \oplus c_2 \oplus c_3 &= 0 \\
  c_0 \oplus c_1 \oplus c_4 &= 0 \\
  c_0 \oplus c_1 \oplus c_2 \oplus c_5 &= 0
\end{align*}
\]

Here “$\oplus$” is a modulo-2 addition. Parity check equation requires that modulo-2 addition of all the bits in the equation should be equal to 0.

Code constraints are normally expressed in a matrix form.

\[
\begin{bmatrix}
1 & 0 & 1 & 1 & 0 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 \\
1 & 1 & 1 & 0 & 0 & 1
\end{bmatrix}
H
\begin{bmatrix}
  c_0 \\
  c_1 \\
  c_2 \\
  c_3 \\
  c_4 \\
  c_5
\end{bmatrix}
= 
\begin{bmatrix}
0 \\
0 \\
0
\end{bmatrix}
\]

12
2.4. SYSTEMATIC AND NON-SYSTEMATIC CODES

In this matrix $H$ is called parity-check matrix. Where each row of $H$ is a parity-check equation and each column represents bits in the codeword. The codeword constraints can also be written as

\[ c_3 = c_0 \oplus c_2 \]
\[ c_4 = c_0 \oplus c_1 \]
\[ c_5 = c_0 \oplus c_1 \oplus c_2 \]

These constraints can also be written in the matrix form as.

\[
\begin{bmatrix}
    c_0 & c_1 & c_2 & c_3 & c_4 & c_5 \\
\end{bmatrix} = \begin{bmatrix}
    c_0 & c_1 \\
\end{bmatrix} \times \begin{bmatrix}
    1 & 0 & 1 & 1 & 1 & 1 \\
    0 & 1 & 0 & 0 & 1 & 1 \\
    0 & 0 & 1 & 1 & 0 & 1 \\
\end{bmatrix}_{G}
\]

Where $G$ is called generator matrix.

Codeword $C$ is calculated using the equation 2.4.1. Where $u$ contains the message bits.

\[ c = uG \]  \hspace{1cm} (2.4.1)

For a binary message with $k$ message bits and $n$ codewords the generator matrix $G$ is $k \times n$ binary matrix. Additionally, the rate of the code is the ratio $k/n$.

Suppose sender wants to send message “101”. The check bits are computed as follows:

\[ c_3 = 1 \oplus 1 = 0 \]
\[ c_4 = 1 \oplus 0 = 1 \]
\[ c_5 = 1 \oplus 0 \oplus 1 = 0 \]

Hence, the codeword of the message will be $[101010]$. 
2.5. LOW DENSITY PARITY CHECK CODES (LDPC)

A generator matrix $G$ of a code having parity check matrix $H$ can be found by performing Gaussian Jordan Elimination on $H$.

$$H = [A, I_{n-k}]$$ \hspace{1cm} (2.4.2)

Where $A$ is $(n-k) \times k$ binary matrix and $I_{n-k}$ is an identity matrix. So from equation 2.4.2 generator matrix can be defined as.

$$G = [I_k, A^T]$$ \hspace{1cm} (2.4.3)

2.4.2 Non Systematic code

A non systematic code is such that the output message does not contain the input message.

2.5 Low Density Parity Check Codes (LDPC)

Low-Density Parity-Check (LDPC) codes are a class of linear block codes. These codes can be explained by sparse parity check matrix $H$. Sparse parity check matrix is a matrix which contains a small number of non-zero values. This sparseness allows LDPC codes in achieve linear time encoding and decoding complexity.

LDPC codes are generally represented in the bipartite graph form as shown in the Figure 2.4. The right nodes are called check nodes and left nodes are called variable nodes. Whereas, edges between these nodes represent an equation defined in $H$.

LDPC codes are either regular or irregular [11]. The number of 1’s in the columns of $H$ matrix is called variable degree and is denoted as $w_v$. Whereas, the number of 1’s in the rows is called check degree and is denoted as $w_c$. If $w_v$ and $w_c$ are constant, then LDPC is called regular. However, if $w_v$ and $w_c$ are not constant, then it is called irregular LDPC.

2.6 Message Passing Algorithm

Message passing algorithms are also called iterative decoding algorithms because they pass messages with the edge of the Tanner graph [8].
2.6. MESSAGE PASSING ALGORITHM

As can be noted in the Figure 2.5, $f_1$ is formed by the modulo-2 addition of $c_1$, $c_5$ and $c_6$. Furthermore, $f_2$ and $f_3$ are formed by $c_2$, $c_3$, $c_6$ and $c_1$, $c_4$, $c_5$ respectively. Nodes labeled with $f$ are called the **check nodes** and nodes labeled with $c$ are called **bit nodes/variable node**.

Considering BEC where values are either received without any error or are erased [11], each bit node sends a message $M_i$ to the connected check nodes. This message is labeled with $i$ i.e. $M_i$, where $i$ denotes the $i$-th bit of the node. Check node receives ‘0’, ‘1’ or ‘x’. Where ‘x’ denotes that the message is still unknown. If the check nodes receive only one unknown ‘x’ then it will calculate the value of ‘x’ that satisfies the parity check equation. The value is later returned to the bit nodes. The bit node whose value was unknown will change the value according to the new incoming message. This process is repeated until all the unknown values are known.

Suppose a sender transmits codeword $c = [101101]$ through a erasure channel and vector $y = [10xxx1]$ is received with three erasures as shown in Figure 2.6(a)(left).
2.6. MESSAGE PASSING ALGORITHM

In the first step bit nodes connected with first check node are calculated. The first check node is connected with 1st, 5th, and 6th bit node. The message received by the check node will be ‘1’, ‘x’ and ‘1’. Bit nodes connected with this check node has only one unknown (‘x’) and that is for the bit node 5 as shown in Figure 2.6 (a)(right orange). The output message of this edge is:

\[ E_{1,5} = M_1 \oplus M_6 \]
\[ E_{1,5} = 1 \oplus 1 = 0 \]

Check node 2 is connected with bit node 2, 3 and 6 as shown in the Figure 2.6 (b)(left orange). The incoming message to the check node will be ‘0’, ‘x’ and ‘1’. Bit nodes connected with the check node 2 has only 1 unknown value and that is of bit node 3. The output message for this node is:

\[ E_{2,3} = M_2 \oplus M_6 \]
\[ E_{2,3} = 0 \oplus 1 = 1 \]
2.6. MESSAGE PASSING ALGORITHM

Figure 2.6: Decoding using the message passing algorithm for the received string $y = [10\ldots x1]$. Sub-figures show the decision taken in each step.
2.7. LT-CODE

Check node 3 is connected of bit node 1, 3 and 5. The incoming message to the check node has two unknown bits (3rd and 5th) as shown in the Figure 2.6 (b)(right purple). This check node will not be able to calculate the unknown values.

In the second step all the unknown nodes which have received a value will update to the new value. 3rd and 5th bit will be changed to known values. After this step we get:

\[ y = [101x01] \]

Step 1 is repeated until all the unknown bits are recovered. As can be seen, check node 3 is connected to 1st, 4th and 5th bit node. An incoming message to this check node will be ‘1’, ‘x’ and ‘0’ Figure 2.6(c)(orange). The output message for this node is:

\[ E_{3,4} = M_1 \oplus M_5 \]
\[ E_{3,4} = 1 \oplus 0 = 1 \]

Hence message is recovered.

\[ y = [101101] \]

2.7 LT-Code

LT-codes are the first practical implementation of fountain code [5]. It was first introduced by Michael Luby in 1998.

In LT-code message \( M \) is divided into \( K \) equal packets \( T_1, T_2, T_3..T_k \). Where packet \( T_i \) is a fundamental unit which is either transmitted or erased. The LT-code does not have any fixed rate, i.e. they generate limitless encoding symbols from a given data. Furthermore, the decoder retrieves original data from any set of received data. So it does not matter what is the loss model of the channel. Sufficient encoding symbols can be generated and sent so that enough symbols are received at the receiver side [6].
2.7. LT-CODE

At each clock cycle LT-encoder generates $K$ random bits $G_{kn}$. Modulo-2 addition is performed on the packets for $G_{kn} = 1$ and then the resulting packet $t_n$ is sent over the erasure channel. This is shown in the equation 2.7.1.

$$t_n = \sum_{k=1}^{K} s_k G_{kn}$$  \hspace{1cm} (2.7.1)

The set of $K$ random bits that are generated to formulate output packets are specified in the column of the generator matrix $G$ as shown by Figure 2.7.

![Figure 2.7: Generator Matrix. Purple column shows the packets erased by the BEC.](image)

At the receiver end, subset of the original generator matrix is received, while the extra packets are erased by the channel. This matrix can be expressed as a $K \times N$ matrix. The matrix should be in the form where $N$ should be equal or greater than $K$ as shown in the Figure 2.8. The original data can be recovered by using the inverse of the generator matrix $G^{-1}$ as shown by the equation 2.7.2.

$$s_k = \sum_{n=1}^{N} t_n G_{kn}^{-1}$$  \hspace{1cm} (2.7.2)

2.7.1 LT-Encoder

The encoding of LT-Code consists of following steps:
2.7. LT-CODE

![Generator Matrix](image)

Figure 2.8: Generator Matrix at the receiver side. Where \( N \geq K \).

- Choose randomly the degree \( d_n \) of a packet from degree distribution.
- Choose random \( d_n \) different input packets.
- Assign \( t_n \) the value of modulo-2 addition of these packets.

Suppose the message to be transmitted using LT-encoding is

\[
M = [1101]
\]

This message is divided into \( K \) packets of 1 bit each. The first packet is transmitted by performing modulo-2 addition on the packets 3 and 4. Second packet is transmitted by performing modulo-2 addition on the packets 1, 2, 3, and so on. Potentially any numbers of output packets can be generated as shown in Figure 2.9.

2.7.2 LT-Decoder

The task of the LT-Decoder is to retrieve \( s_k \) from \( t=sG \). Where \( G \) is the generator matrix of the graph. The simplest way to decode is by using message passing algorithm. Following steps are followed for decoding:

- Find a check node that has degree 1 (connected to only one packet \( s_k \)). If there is no node available with degree 1, decoding halts.
- Assign the value to \( s_k \) (\( s_k = t_n \)).
2.7. LT-CODE

![LT-Encoder Diagram](image)

- Modulo-2 addition of $s_k$ with all the check nodes that are connected with it.
- Remove the edges connected with $s_k$.
- Repeat the above steps until all the packets are decoded.

If the decoder receives, for example, the packets as shown in the Figure 2.10 (red shows the received symbols and green shows the symbols that are to be recovered). Figure 2.11(a)(left) shows the first iteration of the algorithm. We assign the value of $t_3$ to $s_1$ which is 1. As we have the value of $s_1$, now we remove all the edges (orange) after performing modulo-2 addition of $s_1$ with all the check nodes ($t_2$ and $t_6$) it is connected with.

In the second iteration, the check node $t_6$ has degree 1 with node $s_4$ as shown in Figure 2.11(a)(right). The value of $t_6$ is assigned to $s_4$ which is 1. Afterwards, modulo-2 addition is performed of $s_4$ with check nodes it is connected with ($t_1$ and $t_4$) in order to erase the edges.

As can be seen in the Figure 2.11(b)(left), the third iteration, $t_1$ has a single edge with $s_3$, the value of $t_1$ is assigned to $s_3(0)$. Edges connected to node $s_4$ are also removed after performing modulo-2 addition with the check nodes. Finally, the value of $t_2$ is assigned to $s_2(1)$, as $t_2$ has a single edge.
Figure 2.10: Packet received at LT-Decoder. Rest of packets are deleted by the channel.

Figure 2.11(b)(right) shows that the packet is correctly retrieved by the decoder which was sent by the encoder and all the edges are removed (Figure2.11(c)).

### 2.8 Raptor Code

LT-codes has an issue that they cannot be encoded and decoded in linear time \((K \log_e K)\). For this purpose another class of Fountain code was introduced that achieves encoding and decoding in linear time. These codes are called **Raptor Codes**.

Raptor code was first introduced in 2000/2001 by Amin Shokrollahi. Raptor code is an extension to the LT code [7]. Raptor code uses an outer code/pre-code and concatenates it with weak LT-code to patch the gap in the LT-code[6]. Here the term weak refers to small number of edges used for encoding.

Raptor decoding is performed in two stages. In the first stage, only some fraction of symbols can be recovered by the LT-code. In the second stage, symbols that are not recoverable by the LT-code are retrieved by the outer code. Raptor code uses LT-code with an average degree of 3 [6]. This lower average degree ensures that at the decoder side, the decoding does not halt, at the cost that, certain amount of symbols will not be recovered.
Figure 2.11: Example step by step decoding of LT-Code. Here $K = 4$ and $N=6$. 
2.8. RAPTOR CODE

Let $k$ be the input symbols. *Pre-code* is used to generate *intermediate symbols* $I$. LT-Code, inside Raptor code, uses these intermediate symbols to generate the output message as shown in the Figure 2.12.

The encoding and decoding cost of Raptor code is slightly more from that of the LT-code, because we have to keep into account the space it requires to save the intermediate symbols.

### 2.8.1 Raptor Encoder

Encoding of Raptor code is divided into two steps. The first step is to apply *Pre-code* which is capable of error correction if certain symbols are not received at the decoding side. Tornado codes, truncated LT-code, LDPC code, and others can used as a pre-code in the first step to produce intermediate symbols. These intermediate symbols are then used in the second step by the LT-code as an input to produce the output symbols.

Suppose the following packet has to be transmitted over a BEC using Raptor code.
2.8. RAPTOR CODE

Figure 2.13: Raptor encoding: Intermediate symbols are generated in the first step. Later, they are used by LT-code as an input to generate output.

\[ M = [110] \]

In the first step, intermediate symbols are generated using pre-code (LDPC). The generator matrix \( G \) is

\[
G = \begin{bmatrix}
1 & 0 & 0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 1 & 1 \\
0 & 0 & 1 & 1 & 0 & 1
\end{bmatrix}
\]

The Generator matrix generates the intermediate symbols [110100]. In the second step, intermediate symbols will be used by LT-code as an input to generate output symbols as shown in the Figure 2.13.
2.8.2 Raptor Decoder

Raptor code decoding is also divided into two steps. In the first step, the received packets are decoded using LT-Decoder to generate intermediate symbols. In the second step, pre-code is used to decode and retrieve the original message. At the decoder side, LT-code is initially used to retrieve as much packets as possible as shown in the Figure 2.14(a). However, as can be seen, one of the packet could not be recovered by the LT-decoder. In the second step LDPC decoder retrieves that missing packet as shown in the Figure 2.14 (b).
Figure 2.14: Raptor Decoding: (a) Intermediate symbols are retrieved, but one symbol could not be retrieved. (b) Missing symbol was retrieved by the outer code.
Chapter 3

HW-SW Co-Design of Fountain Code on Zynq platform

In this chapter implementations of two different types of Fountain codes (LT-code and Raptor code) will be discussed. Implementation is performed on Zynq platform, the tools used are Vivado and Vivado HLS. First the results of a software implementation on an embedded platform are discussed. This allows us in evaluating which part of the software can be accelerated in hardware. Furthermore, the method of implementation (flow of the code) is also analyzed.

For software implementation, ARM cortex A9 is used. In this thesis encoding and decoding time of LT-code and Raptor code is analyzed on a single core. Furthermore, hardware is designed by using Vivado HLS tool. The reasons for using Vivado HLS tool for the design of the hardware block are:

- Reduction in design time.
- Ease of implementation.
- C based solution.

The design was verified by using Vivado HLS and results were exported in System C format and viewed on GTKwave.

The model used for all the Fountain codes is BEC. Hence, in this thesis the value of erasure probability $\alpha_e$ is taken to be 0.1.
3.1 LT-Code Encoding on Embedded Processor

The data which has to be sent over a channel is first broken into $K$ equal size symbols. The symbol size $T$ is taken to be 4 bytes and 64 bytes. If the data could not be broken to equal size, then 0s are concatenated at the end such that all the packets have the same size as the intended packet size (4 bytes or 64 bytes). Next, a random number $d_n$ is generated using pseudo random number generator so that $d_n$ distinct packets are selected from $K$ symbols. Modulo-2 addition is performed on all those packets and the final outcome is generated. Encoder sends this output packet along with the seed of the pseudo random number generator to the decoder. Figure 3.2 shows the flow chart of LT-encoder.

![Time consumed by an LT-encoder to encode for different data size](image)

Figure 3.1: Time consumed by an LT-encoder to encode for different data size.

The result of the implementation can be seen in Figure 3.1. As it can be seen that the smaller the packet size the more time it will take to encode. The $x$-axis and the $y$-axis show the total data size and the time taken to
3.1. LT-CODE ENCODING ON EMBEDDED PROCESSOR

Figure 3.2: Flow chart of LT-encoder on embedded processor.
encode respectively. Here, total packets which are encoded are $N + \text{packet lost in the erasure channel}$. Where N is $K + 0.1K$. The energy it consumes to encode on the embedded processor (Zynq platform) is shown in the Figure 3.3. Where $x$-axis and the $y$-axis represents the total data size and the energy consumed to encode respectively.

![Figure 3.3: Energy consumed by an LT-encoder to encode for different data size.](image)

3.2 LT-Code Decoder on Embedded Processor

Decoding is the most time consuming part in Fountain code. The decoder is implemented on the embedded processor by following the steps shown in the flow graph of Figure 3.4.

The decoder receives a pair of an output symbol and a seed from the encoder. These received pairs are used to generate a bipartite graph, that is later used in decoding.
The flow graph shown in the Figure 3.4 explains the implementation of the decoder. The decoder waits for $N$ symbols until it starts decoding. As soon as it receives $N$ packets, it checks for packet ($S_k$) with degree 1. After all such packets are found, it searches for all other packets that have $S_k$ as their member. The value of $S_k$ is removed from all of them by performing modulo-2 addition (modulo-2 addition of the same number twice results in 0). The decoder then checks that if all the packets are decoded. Finally, all the packets will contain only one member. If that is the case it stops here, else the steps are repeated.

Figure 3.5 shows the result of the decoder. It can be noted that the decoding time increases exponentially with increasing number of packets. Here $x$-axis and $y$-axis represents number of packets $K$ used for encoding.
3.2. LT-CODE DECODER ON EMBEDDED PROCESSOR

![Graph 1](image1)

Figure 3.5: LT-code decoding. Where K represents number of packets of original message. Here T = 4 bytes and time consumed to decode respectively. Here the size of single packet T is 4 bytes. Secondly, it can also be noted that for the same block size decoding takes far more time than encoding.

![Graph 2](image2)

Figure 3.6: LT-code decoding. Here T = 64 bytes

If the size of a single packet (T) is increased from 4 bytes to 64 bytes, the decoding time decreases. For example, consider a message with 32KB. When T = 4 bytes, K will be 8192 and the time to decode is 13 seconds. However, when T = 64 bytes, K will be 512 and the time for decoding is 0.84 seconds.
This shows that decoding time strongly depends on the number of packets. Additionally, it can also be noted that the decoding time when $T$ is 4 bytes or 64 bytes is almost same if both have the same number of packets to decode (see Figure 3.5 and 3.6). Moreover, the maximum number of symbols feasible to decode is 4096, because for $K > 4096$ latency becomes a bottleneck.

![Figure 3.7: Energy Consumption of LT-code decoding. Here $T = 4$ bytes](image)

Figure 3.7 and Figure 3.8 show the total energy consumption for varying values of $K$ when $T = 4$ bytes and 64 bytes respectively. For achieving a better energy profile a larger packet size $T$ is a better option for a decoder as can be seen from the figures.
3.3 Raptor Code Encoding on Embedded Processor

Raptor code encoding is basically divided into two steps. First, the encoder performs encoding using the pre-code and then the output of the pre-code stage is used by the weak LT-code as an input to generate output. To analyze the implementation, packet size $T$ is chosen to be 4 bytes.

Figure 3.9 shows the flow graph of Raptor code. Where Initially, the data $M$ is divided into $K$ packets. Next, the number of extra packets ($S_{ep}$) are found. A random number $x_n$ is generated using pseudo random number generator. After that $x_n$ distinct packets are selected from $K$ symbols. Modulo-2 addition is performed on all those packets and intermediate node is generated. The process is repeated until $S_{ep}$ number of intermediate nodes are generated. Afterwards these intermediate nodes are passed to the weak LT-code encoder stage. The LT-code encoder is similar to the encoder described in subsection 3.1. However, in this case the average degree $d_n$ of the weak LT-code encoder is 3. This lower degree ensures that encoding and decoding is performed in linear time.
3.3. RAPTOR CODE ENCODING ON EMBEDDED PROCESSOR

![Flow chart of Raptor-encoder.](image)

Figure 3.9: Flow chart of Raptor-encoder.
The number of intermediate nodes are found by the sum of the equation 3.3.1 and 3.3.2. The number of intermediate packets $S_{ep}$ for different blocks sizes is given in table 3.1 [3].

$$E_{ep} \geq \lceil (0.01 \times K) \rceil + X$$  \hspace{1cm} (3.3.1)

$$(H, \text{ceil}(H/2)) \geq K + S$$  \hspace{1cm} (3.3.2)

Where, $X$ is the smallest positive integer such that $X(X-1) \geq 2K$. Additionally $H$ and $E_{ep}$ are the smallest integer and prime integer respectively.

<table>
<thead>
<tr>
<th>Block Size</th>
<th>128</th>
<th>256</th>
<th>512</th>
<th>1024</th>
<th>2048</th>
<th>4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Extra Packets ($S_{ep}$)</td>
<td>29</td>
<td>40</td>
<td>53</td>
<td>72</td>
<td>103</td>
<td>152</td>
</tr>
</tbody>
</table>

Table 3.1: Extra packets generated by the encoder of Raptor code.

Once encoder has generated intermediate packets, i.e. original packets + extra/redundancy node, they are used by an LT-encoder to generate output packets. Here, the total packets that are encoded equals $N + \text{packet lost in the erasure channel}$, $N$ is $K + 10\%$ of $K$. The packets that are lost in the erasure channel are compensated by sending extra packets during the encoding stage. Additionally, the encoder sends a seed of the pseudo random generator along with the packet for the decoder.

Figure 3.10 shows intermediate packets encoding time on embedded processor. Figure 3.11 shows the time for encoding the packets. It shows approximately linear trend between the number of packets and the time consumed to encode them.
3.3. RAPTOR CODE ENCODING ON EMBEDDED PROCESSOR

Figure 3.10: Time consumed to generate intermediate packets by Raptor encoder.

Figure 3.11: Time consumed to generate output packets by Raptor encoder.
3.4 Raptor Code Decoding on Embedded Processor

The decoder decodes in two steps. The decoder receives an output symbol along with the associated seed from the encoder. This information is used to generate a bipartite graph, that is later used in decoding. The decoder waits till it gets $N$ packets. Initially, the decoder uses LT-code decoding to decode as many symbols as possible. Afterwards the pre-code finds the remaining symbols. The decoder uses the seeds for the pseudo random generator to get information about the packets associated with the incoming message.

Figure 3.14 shows the flow graph of decoding. Decoder waits for $N$ symbols to start decoding. As soon as it receives $N$ symbols, LT-decoder in Raptor code finds a packet ($S_k$) which has a degree 1. Then it checks for the received messages which has $S_k$ as its member, removes $S_k$ from all the packets by performing modulo-2 addition. Finally, when all the received messages are decoded, LT-decoder stops.

After LT-decoding stage there remain packets that are not retrieved. Hence the pre-code decoding stage is initiated. The pre-code decoding first checks for missing node (x). It then checks that which intermediate node has this unknown node as its member. It passes all the known values of the node to calculate the unknown value using message passing algorithm (see subsection 2.6). Finally, the value obtained after the calculation is assigned to unknown node.

The result of computation time for Raptor decoding is shown in Figure 3.12 and energy consumed by the decoder can be seen in Figure 3.13. It is found that LT-decoding takes most of the time and energy in Raptor Decoding.
3.4. RAPTOR CODE DECODING ON EMBEDDED PROCESSOR

Figure 3.12: Raptor code decoding time consumption on embedded platform.

Figure 3.13: Raptor code decoding energy consumption on embedded platform.
3.4. RAPTOR CODE DECODING ON EMBEDDED PROCESSOR

Figure 3.14: Flow chart of Raptor decoder.
3.5 Hardware Software Partitioning of LT-decoder

LT decoding consumes more than 90% of the total computation time. Hence, this part of the code is designed in the hardware to increase the performance of the overall code.

The results of hardware accelerator for the LT-code can be seen in Figure 3.15. The hardware block has a maximum frequency of 350 MHz. Additionally, power of the hardware block is 1.797W. Energy consumed by the hardware can be seen in Figure 3.16. It is noted that significant hardware acceleration could not be achieved because of memory accesses. It is concluded that memory access time is more than the computation time for the LT-decoder.

![Graph showing decoding time of LT-decoder using hardware accelerator.](image)

Figure 3.15: Decoding time of LT-decoder using hardware accelerator.

The resources utilized by the hardware are 13% of the FlipFlops, 20% of the LUTs and BRAM used are 8% of the total resources available in the Zynq platform as shown in the Figure 3.17.
3.5. HARDWARE SOFTWARE PARTITIONING OF LT-DECODER

Figure 3.16: Energy consumed by the hardware accelerator.

Figure 3.17: Resourced utilized by the hardware.
Chapter 4

Results and Discussion

LT-code encoder and decoder were implemented with two different packet sizes i.e. 4 bytes and 64 bytes. It was analyzed, that packet size has a pronounced impact on the computation time. When the same data was divided into $K$ numbers of packet and each packet size was 64 bytes, considerably less time was consumed to encode and decode compared with 4 bytes packet size. However, for same number of packets $K$ the encoding and decoding time almost remains the same. Hence, it is proved that encoding and decoding computation time increases as packets $K$ are increased.

Secondly, the maximum number of packets suitable for encoding and decoding on embedded platform is less than or equal to 4096 because latency can become a bottleneck for higher values of $K$.

Thirdly, LT-decoding process consumes more than 90% of the total processing time. Hence, the decoder was implemented in hardware. The results suggest that hardware does not achieve significant improvement in computation time. Memory accesses is the prime hindrance for not achieving considerable improvement in computation time. However, much better performance can be achieved if faster memories are used.

Todor Mladenov et al [3] [4] proposed that they achieved 5.9 times better computation in favor of hardware while requiring 5.5 times lower energy consumption compared to software implementation. However, in the implementation of this thesis 4.6 times better computational performance is
achieved, whereas, energy consumption of hardware implementation is lower than that of a software by a factor of 4.
Chapter 5

Conclusion and Future work

The main objective of the thesis was to evaluate performance of Fountain codes on an embedded platform. For this purpose performance of two types of Fountain codes, namely LT-code and Raptor code are evaluated. The thesis present two scenarios. Firstly, a software implementation of Fountain code is realized and secondly, hardware software partitioning is performed in order to accelerate the LT-decoding stage.

It was found that LT-decoding consumes more than 90% of the time for both implementations (LT-code is also part of Raptor code). Furthermore, it was noted that decoding time increases significantly with the increase in the number of packets. Using a larger symbol size $T$ leads to significant decrease in computation time, and consequently reduces energy consumption. To show this result LT-code was implemented with packet size 4 bytes and 64 bytes. However, having the same number of total packets ($K$) with different packet sizes ($T$) resulted in almost the same encoding and decoding computation time and energy consumption.

A dedicated hardware was presented for the LT-decoder to analyze the differences with its software counterpart. It was noted that hardware performs better than software. However, significant improvement was cannot be achieved as algorithm requires considerable amount of memory accesses.

Accessing memory during computation hampers the advantages that could be achieved by the hardware. Hardware accelerator using caches and fast
memories can be considered to find faster solutions for Fountain code in future.
Bibliography


