4-18-2008

Study of FPGA Implementation of Entropy Norm Computation for IP Data Streams

Subramanya Nagalakshmi

University of South Florida

Follow this and additional works at: https://scholarcommons.usf.edu/etd

Part of the American Studies Commons

Scholar Commons Citation
https://scholarcommons.usf.edu/etd/421

This Thesis is brought to you for free and open access by the Graduate School at Scholar Commons. It has been accepted for inclusion in Graduate Theses and Dissertations by an authorized administrator of Scholar Commons. For more information, please contact scholarcommons@usf.edu.
Study of FPGA Implementation of Entropy Norm Computation for IP Data Streams

by

Subramanya Nagalakshmi

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Engineering
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Co-Major Professor: Srinivas Katkoori, Ph.D.
Co-Major Professor: Rahul Tripathi, Ph.D.
Hao Zheng, Ph.D.

Date of Approval
April 18, 2008

Keywords: IP anomaly, reconfigurable hardware, randomized algorithm, hardware design, traffic analysis

© Copyright 2008, Subramanya Nagalakshmi
DEDICATION

For Putta ...
ACKNOWLEDGEMENTS

I would like to thank Dr. Srinivas Katkoori and Dr. Rahul Tripathi for giving me an opportunity to work on this thesis. They provided me with the initial key references from which to begin the work and to carry out this study. They have guided me throughout the thesis.

I would like to thank Mr. Joe Rogers of Academic Computing, University of South Florida for providing me with the LAN capture data which I could use, to run some my simulations on. I would also like to thank Mr. Peter Schiavo and Mr. Daniel Prieto of CSE Technical Support for their help.
# TABLE OF CONTENTS

LIST OF TABLES iii  
LIST OF FIGURES iv  
LIST OF CODES v  
ABSTRACT vi  

CHAPTER 1 INTRODUCTION 1  
1.1 Thesis Motivation 4  
1.2 Thesis Organization 5  

CHAPTER 2 RANDOMIZED ALGORITHM FOR ESTIMATING ENTROPY OF DATA STREAMS 6  
2.1 Estimating the Entropy Norm 6  
2.1.1 Estimating the Number of Counters $s_1$ and $s_2$ 7  
2.1.2 Simulation Results for Entropy Norm Computation 8  

CHAPTER 3 REDUCING COUNTERS IN ENTROPY CALCULATION ALGORITHM 11  
3.1 Probabilistic Analysis Tools 11  
3.1.1 Chebyshev’s Inequality 11  
3.1.2 Chernoff Bound on Median Estimator 12  
3.1.3 New Reduced Estimates for Counter $s_2$ 13  
3.1.4 Simulation Results with Reduced Counters 14  
3.2 Useful Inequalities for Tightening the Variance Bound 15  
3.2.1 Bound on $X$ 15  
3.2.2 Bound on Sum of Logarithms 16  
3.3 Bound on Variance of the Random Variable $X$ 18  

CHAPTER 4 HARDWARE IMPLEMENTATION IN XILINX FPGA 21  
4.1 Mapping the Algorithm into Hardware 22  
4.2 Design 1: Counter Module 23  
4.2.1 Design 1: Operation 23  
4.2.2 Design 1: Advantages and Disadvantages 26  
4.3 Design 2: Counter Module Using the Block SelectRAM(bRAM) 29  
4.3.1 Design Considerations 29  
4.3.2 Design 2: Advantages and Disadvantages 30
4.4 Logarithm Module  
   4.4.1 Design of the Log Circuit 33  
   4.4.2 Log Circuit Performance 35  
4.5 RNG Module 36  
4.6 Integrated Module 38  
   4.6.1 Hardware and Software Comparisons of the Integrated Module 45  
4.7 Median Module 52  

CHAPTER 5 CONCLUSIONS 53  
5.1 Alternative Design Considerations 55  
5.2 Future Work 56  

REFERENCES 58
LIST OF TABLES

Table 1.1  Anomalies and Packet Features to be Monitored [5]  3
Table 2.1  $F_H$ Computation Simulations - Random Traffic  9
Table 2.2  $F_H$ Computation Simulations - Skewed Traffic  10
Table 3.1  $F_H$ Computation Simulations with Reduced Counters - Random Traffic  14
Table 3.2  $F_H$ Computation Simulations with Reduced Counters - Skewed Traffic  14
Table 4.1  Accuracy and Speed of the Log Circuit in Comparison with Matlab  36
Table 4.2  Comparison of Hardware Simulations Vs Software Simulations - Part 1  47
Table 4.3  Comparison of Hardware Simulations Vs Software Simulations - Part 2  48
Table 4.4  $F_H$ Computation Comparison - One Integrated Module  49
Table 5.1  Device Utilization Summary of FPGA Synthesis on XC2VP30  55
Table 5.2  Summary of Design Capabilities  56
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.1</td>
<td>Upper Bound, True Value, and Lower Bound of $\frac{X}{m}$</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Bound and True Value of Sum of Logarithms</td>
<td>17</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Bound and True Value of Sum of Squares of Logarithms</td>
<td>17</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>A Counter Module</td>
<td>24</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Initial Set up for 16 Counter Modules Simulation Waveforms</td>
<td>27</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>Counters in Operation Simulation Waveforms</td>
<td>28</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Counter Module with bRAM Simulation Waveforms</td>
<td>31</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>Logarithm Module Simulation Waveforms for Computing $\log_2(10^6)$</td>
<td>35</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Matlab Simulation of Random Number Generator</td>
<td>37</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>RNG Module Simulation Waveforms</td>
<td>37</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>The Integrated Module</td>
<td>38</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>Pre-epoch Phase: Initialization</td>
<td>40</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>Epoch Phase: a</td>
<td>41</td>
</tr>
<tr>
<td>Figure 4.11</td>
<td>Epoch Phase: b</td>
<td>42</td>
</tr>
<tr>
<td>Figure 4.12</td>
<td>Epoch Phase: c</td>
<td>43</td>
</tr>
<tr>
<td>Figure 4.13</td>
<td>Post-epoch Phase: Random Variable Computation and Averaging</td>
<td>44</td>
</tr>
<tr>
<td>Figure 4.14</td>
<td>Block Diagram of all 112 Modules and their Interconnect</td>
<td>51</td>
</tr>
<tr>
<td>Figure 4.15</td>
<td>Median Module Simulation Waveforms</td>
<td>52</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>-------</td>
<td>-------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>4.1</td>
<td>Pseudocode of Algorithm for Logarithm Computation</td>
<td>34</td>
</tr>
</tbody>
</table>
Study of FPGA Implementation of Entropy Norm Computation for IP Data Streams

Subramanya Nagalakshmi

ABSTRACT

Recent literature has reported the use of entropy measurements for anomaly detection purposes in IP data streams. Space efficient randomized algorithms for estimating entropy of data streams are available in the literature. However no hardware implementation of these algorithms is available. The main challenge to software implementation for IP data streams has been in storing large volumes of data, along with, the requirement of high speed at which they have to be analyzed. In this thesis, a recent randomized algorithm available in the literature is analyzed for hardware implementation. Software/hardware simulations indicate it is possible to implement a large portion of the algorithm on a low cost Xilinx Virtex-II Pro FPGA with trade-offs for real-time operation. The thesis reports on the feasibility of this algorithm’s FPGA implementation and the corresponding trade-offs and limitations.
CHAPTER 1
INTRODUCTION

The Internet has become the most popular means of communications in every aspect, from simple information exchange to economic activity and transfer of personal and sensitive data. It is economical, viable and the fastest medium for information exchange for a number of everyday activities. But it has also become an easy target for fraud, theft and other forms of attacks. With increased security risks, it has become crucial to monitor and classify any anomalous IP traffic behavior. The consequences of attacks on the Internet can have serious economical impact to the Internet service provider or the organization under attack. The quicker we become aware of an attack, the faster we can respond to safeguard stakeholder interests. One way to monitor the IP traffic involves processing and analyzing these data streams. While software analyzers are popular among network administrators they rely on capture of the IP data stream information in real-time and posteriori off-line analysis of the captured data using various rules [1, 2]. For a 10 Gbit/sec router, with an assumed average packet size of 300 bytes (backbone network [3]), a one minute capture of packet headers at peak load can generate roughly 4.6 GB of data and the analysis thereof can take considerable time. Even on a slower router of 1 Gbit/sec and average packet size ranging from 50 bytes (Email type traffic) to 1500 bytes (Ethernet MTU– maximum transfer unit), 2.8 GB to 95 MB respectively, of data needs to be captured in 1 minute and analyzed. Thus valuable time that could be used to respond to an attack is often lost and large storage requirements are an essential feature of such software analyzers. It has also become important to design some means for summarizing network traffic information in a succinct way, so that one may perform useful analysis (for example: detecting denial of service attacks, detecting spam broadcast) of a network stream in real-time. Additionally,
it makes sense to maintain summary of network traffic at dedicated points that may be used at any time to analyze traffic flow across two or more points [4].

The measurement of anomalous behavior in Internet traffic using packet header features such as addresses, port numbers through Shannon’s entropy has been a growing research area in recent years. Entropy has been shown to be a sensitive measure for detecting anomalies [5]. It has also been shown to present lots of challenges [6]. Automating the process of detecting anomalies is quite challenging because of the difficulty in dividing the traffic into what is normal and what is anomalous behavior [5]. Variations in the traffic flow might be small enough to be considered anomalous at a local level but at a macro level it can be drowned as insignificant. By using entropy calculations on packet destination/source addresses and destination/source port numbers as features in a higher level principal component analysis algorithm, authors in [5] show success in sensitive classification between normal and anomalous behavior. In [8], while the entropy based features are used, the principal component analysis algorithm is shown to need considerable tuning to provide good classification between normal and anomalous behavior. General consensus in the research community seems to be that entropy of features of packet headers is a sensitive measure for use in higher level algorithms to separate normal and anomalous traffic. In Table 1.1, the entropy of features to be monitored for successful detection of various types of attacks from [5] is reproduced.

For IP traffic data, the Shannon entropy of a packet feature which takes \(1, \ldots, n\) different values with corresponding discrete probabilities \(p_i\) is given by

\[
H = - \sum_{i=1}^{n} p_i \log_2 (p_i) \leq \log_2(n) \quad (1.1)
\]

The inequality in (1.1) is exact when the feature is uniformly distributed. Computing entropy requires the probabilities \(p_i\). A simple approach to estimating the probabilities involves the computation of a histogram. For example, consider the IP port numbers ranging from 0 to 65535 which are divided into 3 categories by [7] as Well Known Ports (0 to 1023), Registered Ports (1024 to 49151), and Dynamic and/or Private Ports (49152
Table 1.1. Anomalies and Packet Features to be Monitored [5]

<table>
<thead>
<tr>
<th>Anomaly type</th>
<th>Explanation</th>
<th>Packet features to check</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alpha flows</td>
<td>Usually large volume P2P</td>
<td>Source &amp; destination address.</td>
</tr>
<tr>
<td>DOS</td>
<td>Denial of Service Attack (distributed or single-source)</td>
<td>Destination &amp; source address.</td>
</tr>
<tr>
<td>Flash Crowd</td>
<td>Unusual burst of traffic to single destination, from a typical distribution of sources</td>
<td>Destination address &amp; port.</td>
</tr>
<tr>
<td>Port Scan</td>
<td>Probes to many destination ports on a small set of destination addresses</td>
<td>Destination address &amp; port</td>
</tr>
<tr>
<td>Outage events</td>
<td>Traffic shifts due to equipment failures or maintenance</td>
<td>Source &amp; destination address</td>
</tr>
<tr>
<td>Point to Multipoint</td>
<td>SPAM broadcast</td>
<td>Source &amp; destination address</td>
</tr>
<tr>
<td>Worms</td>
<td>Malicious code broadcast</td>
<td>Destination address &amp; port</td>
</tr>
</tbody>
</table>

to 65535). In [9] packets with a destination port in the Well Known ports are divided into bins of 10 port numbers each. Since packets with port number 80 comprise the majority of the network traffic, they are separated in [9] into a single bin. This produces 104 packet bins. Packets with destination port in the Registered Ports are divided into 482 additional bins, with each bin covering 100 port numbers with the exception of the bin that covers the last 28 port numbers from 49124 to 49151. Packets with Dynamic and/or Private Port numbers are grouped into a single bin. The captured IP data can now be analyzed using the 587 different bins and the frequencies of each bin divided by the total number of packets analyzed is taken to represent the probabilities $p_i$ needed to compute the entropy in (1.1). While entropy calculations are simple to conceptualize, the direct approach of histogram computation and estimating probabilities from it suffers from the same drawback as capture of data in current software packages viz. large memory requirements and slow analysis. Using such entropy features in higher level algorithms, therefore, also suffers. It should be noted that while port number range is known apriori, the range of packet addresses in a stream of IP traffic cannot be known apriori and therefore dynamic bin allocation for histogram computation on packet addresses may be needed, adding to the complexity of such schemes.
Recently, a randomized algorithm [13] that uses less memory in computing entropy on data streams has been proposed. The data stream view of Internet traffic is aimed at avoiding capture of the packet header data. Instead, as each packet header arrives decisions are made immediately and processing of that packet is completed before the next packet arrives. Thus all such algorithms are single pass in nature. The randomized algorithm proposed for entropy calculations on data streams has been studied on data collected from routers [14] and shown to be successful in computing the entropy of the selected feature in an off-line software analysis. The algorithm uses a large number of independent identically distributed random variables whose arithmetic averages in groups of $s_1$ are computed first and then the median over $s_2$ of these groups of the arithmetic average is computed to estimate a measure of the entropy. The variance of the final estimate is greatly reduced by this two stage process.

1.1 Thesis Motivation

To provide a reasonable assessment of attacks requires monitoring of IP traffic for reasonable time periods (epochs). Furthermore, if the intent is to use the detected anomalies for real-time control of the traffic and stop an attack, then the analysis has to be rapid. Currently, most studies in the research literature are posteriori analysis of anomalies on data collected from routers and the run-time of the software is high, and the schemes are not conducive for near real-time application. To be useful, such anomaly detection schemes have to keep up to the high data rates with the analysis throughput comparable to epochs of short duration so that response measures can be activated sensitively.

While entropy calculations are simple if the requisite probabilities are known, and it has been shown to be sensitive to detect anomalies, the direct approach of histogram computation and estimating probabilities suffers from the same drawback as capture of data in current software packages viz. large memory requirements and slow analysis. The randomized algorithm of [13] uses less memory in computing entropy. This algorithm has been studied on data collected from routers [14] and shown to be successful in computing
the entropy of the selected features in an off-line software analysis. No attempt has been
made to implement this algorithm in hardware in the literature. Dedicated hardware for
such applications should be several orders of magnitude faster and thus may be conducive
to the use of entropy for real-time analysis. This thesis is a feasibility study of implementing
some of the core parts of the randomized algorithm for entropy calculation in hardware,
using the Xilinx FPGA tools.

1.2 Thesis Organization

In Chapter 2, the randomized algorithm of [13] that is studied in this thesis will be
discussed and simulation results of the algorithm in Matlab will be presented. The limita-
tions of the algorithm for hardware implementation will be pointed out. In Chapter 3, a
new derivation for the number of counters needed for the algorithm of [13] is carried out
to reduce the overall number of counters. An attempt at tightening the variance bounds
derived in the literature for the algorithm is also discussed. In Chapter 4, the hardware
design of the counter modules is considered and design trade offs are explored and sim-
ulated, and the results presented. A logarithm computation circuit along with averaging
and median calculating circuitry needed for post-epoch processing in the algorithm are also
designed and studied. The designs are integrated into a module. Chapter 5 concludes the
thesis with some possible alternative design strategies and suggestions for future work.
CHAPTER 2
RANDOMIZED ALGORITHM FOR ESTIMATING ENTROPY OF DATA STREAMS

A number of results using entropy as a measurement for anomaly detection purposes are available in literature [5, 9, 10, 11]. This thesis focuses on the randomized algorithm for entropy and entropy norm estimation based on the work available in [13] which has been further studied by [14]. In [13] the metric called entropy norm \( F_H \) as defined in (2.1) is estimated.

\[
F_H = \sum_{i=1}^{n} m_i \log_2(m_i) \quad (2.1)
\]

In (2.1) it is assumed that there are \( n \) distinct packet headers (packet feature) \( 1, \ldots, i, \ldots n \), and for each \( i \), the packet header repeats \( m_i \) times in the stream. The total number of packets in the stream is \( \sum m_i = m \). If the probability of item \( i \) is taken as \( \frac{m_i}{m} \), the entropy norm defined in (2.1) can be related to the entropy of the data stream of \( m \) as follows:

\[
H = \sum_{i=1}^{n} \frac{m_i}{m} \log_2 \left( \frac{m}{m_i} \right) = \log_2(m) - \frac{F_H}{m} \quad (2.2)
\]

The key contribution of [13] is that the algorithm for estimating \( F_H \) has sub-linear space requirements from which the entropy \( H \) can be reasonably estimated to within a \( (1 + \varepsilon) \) factor.

2.1 Estimating the Entropy Norm

In [13], the authors introduce an estimator \( Y \) defined in (2.3), a random variable, which gives the estimate for \( F_H \). The algorithm uses \( s_1, s_2 \) independent basic estimators \( \{X_{ij} : 1 \leq i \leq s_1, 1 \leq j \leq s_2\} \), where each \( X_{ij} \) is an independently, identically distributed
random variable, as the random variable \( X \) defined below in Code 2.1. The proof that the random variable \( Y \) estimates \( F_H \) to within a \((1 \pm \varepsilon)\) factor with a (confidence limit) probability greater than \((1 - \delta)\) is shown in [13]. The choice of values for the variables \( s_1, s_2 \) which represent the number of counters needed to compute \( Y \) is based on \( \varepsilon, \delta \) as discussed below. It is also shown in [13] that individual random variable \( X \) is such that \( E(X) = F_H \). Thus the repeated averaging of the random variables \( X_{ij} \) over \( s_1 \) counters reduces the variance of the estimated \( F_H \) while the median computation over the subsequent \( s_2 \) averages reduces the variance further and ensures the confidence limit.

\[
Y = \text{median}_{1 \leq j \leq s_2} \left\{ \frac{1}{s_1} \sum_{i=1}^{s_1} X_{ij} \right\} \quad (2.3)
\]

---

Input Stream: \( A = \langle a_1, a_2, \ldots, a_m \rangle \), where each \( a_i \in \{1, 2, \ldots, n\} \).
1. Choose \( p \) uniformly at random from \( \{1, \ldots, m\} \).
2. Let \( r = |\{ q : a_q = a_p \text{ and } p \leq q \leq m \}| \). Here \( r \geq 1 \).
3. Let \( X = m (r \log_2(r) - (r - 1) \log_2(r - 1)) \) with a convention that \( 0 \log_2(0) = 0 \).

Code 2.1. Pseudocode for Computing the Random Variable \( X \) [13]

### 2.1.1 Estimating the Number of Counters \( s_1 \) and \( s_2 \)

For any \( \varepsilon, \delta \in (0, 1] \), in [13] it is shown that if \( s_1 \geq 8 \frac{\text{Var}[X]}{(\varepsilon^2 E(X))^2} \) and \( s_2 \) is chosen as integer \( \geq 4 \log_2(1/\delta) \) then the final estimator \( Y \) in (2.3) deviates from \( E(X) \) by no more than \( \varepsilon E(X) \) with probability at least \( 1 - \delta \). Assuming that \( F_H > \frac{m}{\Delta} \), in [13], it is shown that the bound \( \frac{\text{Var}[X]}{E(X)^2} < |\log_2(e) + \log_2(m) - 1| \Delta = O(\Delta \log_2(m)) \). \( \Delta > 0 \) is assumed in [13] while \( \Delta \) is assumed to be 1 or lower in [14]. The condition \( \Delta = 1 \) implies \( m > 2n \) [14]. Using the bound and \( \Delta = 1 \) the minimum number of counters \( s_1 \geq 8 \) \( |\log_2(e) - 1 + \log_2(m)| / \varepsilon^2 \). For \( \Delta = 1, \delta = 0.2 \) and \( \varepsilon = 0.2 \) with \( m = 10^6 \), the corresponding values of \( s_1, s_2 \) are thus 4075 and 10 respectively. With \( m = 10^5 \), the corresponding values of \( s_1, s_2 \) are thus 3411 and 10 respectively.
2.1.2 Simulation Results for Entropy Norm Computation

Since the number of counters needed for the algorithm is dependent on the quantity \( m \), consideration was given to how many of these could fit on a FPGA and at the same time, not under utilize it. Based on an initial trial and error process, an epoch size \( (m) \) of a million packets was fixed. The algorithm was coded in Matlab to examine its performance. A stream of size \( m = 10^6 \) with \( n \) different integers is generated using Matlab random function to simulate the packets. The number of counters \( s_1, s_2 \) are computed as per the above subsection. Although not possible in practice because the number of distinct packet headers \( (n) \) is not known apriori for a given epoch size \( (m) \), in simulations we have the luxury of computing the histogram of the packets in our epoch. Using this histogram, the true entropy norm can be easily computed in the simulations. In each case, in Table 2.1, the number of counters used is \( s_1 = 4075, s_2 = 10 \), respectively i.e., a total of 40750 counters with \( \varepsilon = 0.2, \delta = 0.2 \) and \( \Delta = 1 \). To study the variability of the algorithm of [13], the same stream is analyzed ten times (with the 40750 counters being selected at random locations in the stream each time). Each run of the algorithm in the Matlab code takes about 14 minutes for the analysis of a million packet stream on a Windows dual core Pentium PC with a clock of 1.54 GHz. The nomenclature used in Table 2.1 where the results of these simulations are reported is as follows:

1. \( n \) = number of distinct packets in the stream. The requirement that \( m > 2n \) is maintained in these simulations.

2. \( F_{hm} \) = Entropy norm of [13] (first run value out of the ten is given)

3. \( F_t \) = Entropy norm computed from the histogram of stream. Note that this computation is possible only in simulations.

4. Percentage error in entropy norm = \( \frac{100|F_{hm} - F_t|}{F_t} \). This is computed for each run and the maximum value over all 10 runs is reported.
5. The ratio $F_{hm}/F_t$ should lie within 0.8 to 1.20 (in at least 80% of the simulation runs). This is reported as the minimum, average, and maximum over all 10 runs.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$F_{hm}$</th>
<th>$F_t$</th>
<th>% error in norm</th>
<th>ratio $F_{hm}/F_t$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500</td>
<td>$9.3763 \times 10^6$</td>
<td>$9.3820 \times 10^6$</td>
<td>0.1666</td>
<td>0.9983, 0.9998, 1.0009</td>
</tr>
<tr>
<td>10000</td>
<td>$6.6476 \times 10^6$</td>
<td>$6.6512 \times 10^6$</td>
<td>0.1895</td>
<td>0.9984, 0.9996, 1.0019</td>
</tr>
<tr>
<td>50000</td>
<td>$4.3501 \times 10^6$</td>
<td>$4.3586 \times 10^6$</td>
<td>0.3529</td>
<td>0.9965, 1.0002, 1.0032</td>
</tr>
<tr>
<td>500000</td>
<td>$1.4110 \times 10^6$</td>
<td>$1.4111 \times 10^6$</td>
<td>1.2181</td>
<td>0.9878, 1.0005, 1.0068</td>
</tr>
</tbody>
</table>

The above simulations were carried out at a higher value of $\varepsilon = 0.2$ because, the number of counters $s_1$, depends on this value. A smaller value of $\varepsilon = 0.1$ increases the counters by a factor of 4 times. In spite of the $\varepsilon = 0.2$ and $\delta = 0.2$, since the errors in computing the entropy norm is very small, it is quite clear that the bounds given in [13] are quite loose if the traffic is considered random. Random traffic pattern implies every packet header up to the number being considered ($n$) is equally likely to occur in the stream. Roughly, each packet header then repeats $m/n$ times in the stream. To study the situation of traffic which is somewhat more skewed as is likely in an attack type situation, another set of simulations has been considered. In these simulations the stream is generated from two vectors $q, f$ where each element of $q$ is the number of individually distinct packet headers and the corresponding element of $f$ is their frequency of repetition. Thus $q = [1, 40700, 9299], f = [9301, 2, 1]$ implies that the first packet header is repeated 9301 times, 40700 distinct successive packet headers are each repeated 2 times and the last 9299 distinct packet headers are repeated just once. The total number of distinct packet headers is $n = \sum_{i=1}^{3} q_i = 5 \times 10^4$, while the total number of elements in the stream is $m = \sum_{i=1}^{3} q_i f_i = 10^5$. The randomness in the stream is now in the location of the particular packet header whose repetitions are specified in $q, f$ in the overall stream of length $m$. Two cases are considered in Table 2.2 with the corresponding $q, f$ as given below: Case 1: $q_1 = [1, 40700, 9299], f_1 = [9301, 2, 1]$. Case 2: $q_2 = [5, 10, 100, 2000, 47885], f_2 = [4823, 1000, 100, 4, 1]$. A stream described by vectors $q, f$ is generated in Matlab and the generated stream analyzed by
the algorithm. Each of the two cases has \( m = 10^5, n = 5 \times 10^4 \) and the other algorithm parameters are \( \varepsilon = 0.2, \delta = 0.2 \) and \( \Delta = 1 \). On each stream (Case), the algorithm is run ten times. The corresponding \( s_1, s_2 \) used for each run are 3411, 10 respectively and the total number of counters for these cases is \( s_1 \times s_2 = 34110 \). The results of these simulations are presented in Table 2.2 with similar nomenclature. It is seen from Tables 2.1, 2.2 that in all cases, the algorithm seems to provide a better estimate than prescribed by \( \varepsilon, \delta \) choice. This is largely the result of overestimation of the variance bound and therefore subsequently the number of counters needed by the algorithm. For an epoch of a million packets, as will be seen later, the overall count of 40750 counters is quite above the number of counters implementable in the available FPGA. While \( \varepsilon, \delta \) can be tightened, the drawback is that this would increase the number of counters further. This is the limitation of the algorithm given in [13]. There have been no further improvements to these bounds in the literature. Some attempts to reduce the number of counters in this thesis is documented in the next chapter.

<table>
<thead>
<tr>
<th>Case</th>
<th>( F_{hm} )</th>
<th>( F_l )</th>
<th>% error in norm</th>
<th>ratio ( \frac{F_{hm}}{F_l} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>( 2.0108 \times 10^5 )</td>
<td>( 2.0402 \times 10^5 )</td>
<td>2.3975</td>
<td>0.9760, 0.9966, 1.0154</td>
</tr>
<tr>
<td>Case 2</td>
<td>( 4.8363 \times 10^5 )</td>
<td>( 4.7716 \times 10^5 )</td>
<td>1.4486</td>
<td>0.9991, 1.0053, 1.0145</td>
</tr>
</tbody>
</table>
In this chapter, the counters needed for the algorithm in [13] are reduced by first using a stronger bound on median computation from [12] and then an attempt at tightening the bound on $Var(X)$ is shown. While many of the techniques used in the derivations carried out in this chapter are available in the literature, their particularization to the problem at hand, is new, in this thesis. The basic probabilistic analysis tools on which the algorithm of [13] rests are Chebyshev’s inequality and Chernoff bound for medians [12] and these are discussed next. The goal is to use a stronger result on the Chernoff bound for medians from [12] to obtain a reduction in overall counters.

3.1 Probabilistic Analysis Tools

In this section Chebyshev’s inequality which describes the distribution of a random variable around its expectation is stated. This inequality is useful as it holds independent of the underlying probability distribution function of the random variable. Thereafter, the application of Chernoff bound to median computation based on a theorem in [12] is also provided. These two results and the definition of a basic random variable $X$ are the basis of the randomized algorithm of [13].

3.1.1 Chebyshev’s Inequality

The inequality states for a random variable $X$ with expectation $E(X)$ and variance $Var(X)$, for any $a > 0$

$$
Pr(|X - E(X)| \geq a) \leq \frac{Var(X)}{a^2}
$$

(3.1)
Specifically with \( a = \varepsilon E(X) \), Chebyshev’s inequality states

\[
Pr (|X - E(X)| \geq \varepsilon E(X)) \leq \frac{Var(X)}{\varepsilon^2 E(X)^2}
\] (3.2)

If \( s_1 \) independent, identically, distributed random variables with the same distribution as \( X \) are arithmetically averaged then the random variable \( Y \) has the properties

\[
Y = \frac{\sum_{i=1}^{s_1} X_i}{s_1}, \quad E(Y) = E(X), \quad Var(Y) = \frac{Var(X)}{s_1}
\] (3.3)

Thus by Chebyshev’s inequality (3.2), the distribution of \( Y \) satisfies

\[
Pr (|Y - E(X)| \geq \varepsilon E(X)) \leq \frac{Var(X)}{s_1 \varepsilon^2 E(X)^2} = \frac{1}{K}
\] (3.4)

Thus arithmetical averaging of the random variables ensures that the new estimator \( Y \) has a reduced probability on the spread of its expectation. \( K \) in (3.4) is usually chosen a positive integer (> 2 (from next subsection)). For a given choice of \( K \), \( s_1 \) can be obtained as

\[
s_1 = \frac{KV ar(X)}{\varepsilon^2 E(X)^2}
\] (3.5)

### 3.1.2 Chernoff Bound on Median Estimator

Consider independent identically distributed random variables \( Y_i, i = 1, \ldots, s_2 \) each of which is as described in (3.3). Consider the median of these \( s_2 \) random variables. Let this be denoted \( Z \). \( E(Z) = E(X) \). In this subsection the relationship of \( s_2 \) to \( \delta \) is determined such that

\[
Pr (|Z - E(X)| \geq \varepsilon E(X)) \leq \delta
\] (3.6)
Let random variables $Q$ and $R$ denote the number of $Y_i$ such that $Y_i < (1 - \varepsilon)E(X)$ and $Y_i > (1 + \varepsilon)E(X)$ respectively, then

$$\Pr(|Z - E(X)| \geq \varepsilon E(X)) = \Pr\left(Q \geq \frac{s_2}{2} \text{ or } R \geq \frac{s_2}{2}\right) \leq \Pr\left(Q \geq \frac{s_2}{2}\right) + \Pr\left(R \geq \frac{s_2}{2}\right)$$

(3.7)

$E(Q)$, $E(R)$ are both $\leq \frac{s_2}{K}$. The Chernoff Bound Theorem 4.4(a) of [12] states that for random variables of type $Q$, $R$ the inequalities on probabilities on the right hand side of (3.7) can be written for $\alpha > 0$ (this implies $K > 2$ in (3.8)) as:

$$\Pr\left(Q \geq \frac{K}{2} s_2\right) = \Pr\left(Q \geq (1 + \alpha)E(Q)\right) = \left(\frac{e^\alpha}{(1 + \alpha)^{1+\alpha}}\right)^{E(Q)} = \left(\frac{e^{(1 + 0.5)} - 1}{(0.5K)^{0.5K}}\right)^{\frac{s_2}{K}}$$

(3.8)

Hence (3.7) can be simplified to

$$\Pr(|Z - E(X)| \geq \varepsilon E(X)) \leq 2 \left(\frac{e^{(1 + 0.5)} - 1}{(0.5K)^{0.5K}}\right)^{\frac{s_2}{K}} \leq \delta$$

(3.9)

For a given $\delta$ and $K$, an odd integer $s_2$ can be solved for by using (3.9). For example with $\delta = 0.2$, and for $K = 5, 6, 7, 8$ the corresponding values for $s_2$ are 15, 11, 9, 7 respectively.

### 3.1.3 New Reduced Estimates for Counter $s_2$

For any $\varepsilon, \delta \in (0, 1]$ and integer $K > 2$ specified by the user, if $s_1 \geq K \frac{\text{Var}[X]}{(\varepsilon^2 E(X))^2}$ and $s_2$ is determined as above for the choice of $K$, then the final estimator $Y$ in (2.3) deviates from $E(X)$ by no more than $\varepsilon E(X)$ with probability at least $1 - \delta$. The tighter bound in the above subsection gives the value of $s_2$ as 7 for $K = 8$ with $\varepsilon = 0.2$ and $\delta = 0.2$. While for [13] the corresponding value of $s_2$ is 10. Since each of the $s_2$ estimates are obtained using $s_1$ independent random variables $X$, using a smaller value leads to lesser overall count. For the numerical values illustrated there is a 30% reduction in overall counters ($s_1 \times s_2 = 28525$ instead of 40750). Computing the product $s_1 \times s_2$ for various values of $K$
using the results of the previous subsections, indicates that the optimum choice minimizing the product is at \( K = 8 \) with \( s_2 = 7 \).

### 3.1.4 Simulation Results with Reduced Counters

The same streams used in Chapter 2, Section 2.1.2 are now used to compute the entropy norm with the reduced counters as obtained above with \( \varepsilon = 0.2, \delta = 0.2 \) and \( \Delta = 1 \). The corresponding tables with the same nomenclature as in Tables 2.1 and 2.2 are given in Tables 3.1 and 3.2 respectively. In Table 3.1 28525 random variables (counters) are used for the epoch of \( 10^6 \) packet headers. In Table 3.2, for an epoch of \( 10^5 \) packet headers, a total of 23877 random variables (counters) are used. The two cases in Table 3.2 are considered as in Table 2.2 with the corresponding \( q, f \) as given below: Case 1: \( q_1 = [1, 40700, 9299], f_1 = [9301, 2, 1] \) and Case 2: \( q_2 = [5, 10, 100, 2000, 47885], f_2 = [4823, 1000, 100, 4, 1] \).

Table 3.1. \( F_H \) Computation Simulations with Reduced Counters - Random Traffic

<table>
<thead>
<tr>
<th>( n )</th>
<th>( F_{hm} )</th>
<th>( F_t )</th>
<th>% error in norm</th>
<th>ratio ( \frac{F_{hm}}{F_t} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1500</td>
<td>( 9.3827 \times 10^6 )</td>
<td>( 9.3820 \times 10^6 )</td>
<td>0.1435</td>
<td>0.9986, 1.0000, 1.0014</td>
</tr>
<tr>
<td>10000</td>
<td>( 6.6434 \times 10^6 )</td>
<td>( 6.6512 \times 10^6 )</td>
<td>0.2313</td>
<td>0.9981, 1.0000, 1.0023</td>
</tr>
<tr>
<td>50000</td>
<td>( 4.3724 \times 10^6 )</td>
<td>( 4.3586 \times 10^6 )</td>
<td>0.3929</td>
<td>0.9961, 1.0007, 1.0039</td>
</tr>
<tr>
<td>500000</td>
<td>( 1.4013 \times 10^6 )</td>
<td>( 1.4111 \times 10^6 )</td>
<td>1.6461</td>
<td>0.9835, 0.9963, 1.0118</td>
</tr>
</tbody>
</table>

Table 3.2. \( F_H \) Computation Simulations with Reduced Counters - Skewed Traffic

<table>
<thead>
<tr>
<th>( q, f )</th>
<th>( F_{hm} )</th>
<th>( F_t )</th>
<th>% error in norm</th>
<th>ratio ( \frac{F_{hm}}{F_t} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Case 1</td>
<td>( 2.0796 \times 10^5 )</td>
<td>( 2.0402 \times 10^5 )</td>
<td>2.2569</td>
<td>0.9916, 1.0100, 1.0226</td>
</tr>
<tr>
<td>Case 2</td>
<td>( 4.7938 \times 10^5 )</td>
<td>( 4.7716 \times 10^5 )</td>
<td>1.1306</td>
<td>0.9952, 1.0028, 1.0113</td>
</tr>
</tbody>
</table>

These tables reveal that the reduced counters still produces entropy norm values well within the bounds proscribed by the \( \varepsilon = 0.2 \) and \( \delta = 0.2 \). However with the decreased counters both the errors and the min, max spread of the ratio of the norm estimate to the true norm has increased slightly in most cases. This is only to be expected. It is important to realize that a 30% reduction in the counters still keeps these errors to about 2% even in the worst case while the expectation with the \( \varepsilon = 0.2 \) is that the errors will be
around (within) 20%. This indicates that the variance bound imposed theoretically, based on which the number of counters are derived, is very conservative. An attempt to reduce the variance bound is next developed in the following sections of this chapter.

3.2 Useful Inequalities for Tightening the Variance Bound

In this section, some useful bounds (inequalities) are shown either from the literature or by deriving them.

3.2.1 Bound on \( X \)

The following bound on \( X \) is first shown.

\[
\left( \log_2 e + \log_2 (r - 1) \right) < \frac{X}{m} = \left( r \log_2 r - (r - 1) \log_2 (r - 1) \right) < (\log_2 e + \log_2 (r))
\] (3.10)

Of the two bounds in (3.10), the upper bound is available in [14] while the lower bound is a new derivation. The upper bound follows by considering

\[
2^X \left( \frac{r^r}{(r - 1)^{r - 1}} \right) = r \left( \frac{r}{r - 1} \right)^{r - 1} = r \left( \frac{1}{r - 1} \right)^{r - 1} < e r
\] (3.11)

The last step follows from the fact that for finite \( r > 1 \), \( (1 + 1/(r - 1))^{r - 1} \) is \(< e\) and asymptotically approaches \( e\) as \( r \to \infty\). The lower bound follows by considering

\[
\frac{r^r}{(r - 1)^{r - 1}} = (r - 1) \left( \frac{r}{r - 1} \right)^r = (r - 1) \left( 1 + \frac{1}{r - 1} \right)^{r - 1} \left( 1 + \frac{1}{r - 1} \right) > e(r - 1)
\] (3.12)

The last step follows from the fact that for finite \( r > 1 \), \((1 + 1/(r - 1))^{r - 1}(1 + 1/(r - 1))\) is \(> e\) and asymptotically approaches \( e\) as \( r \to \infty\).

A plot of the inequalities of (3.10) is shown in Fig. 3.1. The inequalities are within 1% of the true value of \( X/m \) for \( r \geq 14 \) and within 5% for \( r \geq 4 \). The right-hand inequality of (3.10) will be exploited later in Chapter 4.
3.2.2 Bound on Sum of Logarithms

A number of useful inequalities involving logarithmic series are derived in this section. The logarithm function is concave, (second derivative is $< 0$). Interpreting the sum in the following logarithmic series as rectangular approximation of the area under the logarithm function, an inequality can be developed as per (3.13).

$$
\sum_{r=2}^{m} \log_{2}(r) < \frac{1}{\ln(2)} \int_{2}^{m+1} \ln(x) dx = \frac{[(m + 1)\ln(m + 1) - 2\ln(2) - m + 1]}{\ln(2)}
$$

$$
= (m + 1) \log_{2}(m + 1) - 2 - (m - 1) \log_{2}(e) \quad (3.13)
$$

The last step in (3.13) is obtained by integration by parts. The upper limit of the integration is $m + 1$ as this is needed to encompass the last rectangle in the series. A plot of the bound and the true values for various values of $m$ is depicted in Fig. 3.2 and the error as $m$ increases is within 2.5% for $m > 22$.

Next, the following sum of squares logarithmic series can also be bounded similarly:

$$
\sum_{r=2}^{m} [\log_{2}(r)]^2 < \left[ \frac{1}{\ln(2)} \right]^2 \int_{2}^{m+1} [\ln(x)]^2 dx
$$

$$
= \frac{[(m + 1)\ln(m + 1)]^2 - 2[\ln(2)]^2 - 2 \int_{2}^{m+1} \ln(x) dx}{[\ln(2)]^2}
$$

$$
= (m + 1) (\log_{2}(m + 1))^2 - 2 - 2 \log_{2}(e) \int_{2}^{m+1} \log_{2}(x) dx \quad (3.14)
$$
In the right hand side of (3.14) the integral left can be evaluated into a closed form expression. This is not shown here as in the typical use of this inequality in this thesis, this integral cancels with another as will be seen in (3.20) the next subsection. A plot of the bound of (3.14) and the true values for various values of $m$ is depicted in Fig. 3.3 and the error as $m$ increases is within 3.5% for $m > 24$. Two conservative inequalities which are used in the following section are next derived.

\[
\frac{\sum_{i=1}^{n} (\log_2(m_i))^2}{\sum_{i=1}^{n} m_i \log_2(m_i)} \leq k = 0.5283 \\
\frac{\sum_{i=1}^{n} \log_2(m_i)}{\sum_{i=1}^{n} m_i \log_2(m_i)} \leq k = 0.5 \quad (3.15)
\]
The approach taken to derive is to assume there exists a bound $k$ for each inequality and then to calculate it conservatively. With $x_i = \log_2(m_i)$ and $x_i = 1$ for the left hand side of the first and second inequality in (3.15), the numerator and denominator on the left hand side can be written as below and with a bound $k$ the implication in (3.16) is obtained:

$$\frac{\log_2 (\prod_{i=1}^n m_i^{x_i})}{\log_2 (\prod_{i=1}^n m_i^{-x_i})} \leq k \Rightarrow \prod_{i=1}^n m_i^{x_i - km_i} \leq 1$$  \hspace{1cm} (3.16)

The inequality in (3.16) can be conservatively satisfied if

$$k \geq \max \left( \frac{x_i}{m_i} \right) \quad m_i = 2, 3, \ldots$$  \hspace{1cm} (3.17)

In (3.17), the value of $m_i$ is $> 1$, since 1 power anything (in (3.16)) is 1 and does not contribute to the bound. A simple computation now reveals that for the first inequality the max in (3.17) is at $k = 0.5283$ when $m_i = 3$ and for the second inequality it is at 0.5 when $m_i = 2$ as claimed in (3.15).

### 3.3 Bound on Variance of the Random Variable $X$

In this section, the variance of the random variable $X$ is bounded. While the section follows the tight bound section derived in [14], the bounding is done using the inequalities derived in the previous sections in an attempt to obtain new insight on the variance.

$$\text{Var}(X) = E(X^2) - (E(X))^2$$  \hspace{1cm} (3.18)

$$E(X^2) = m \sum_{i=1}^n \sum_{r=1}^{m_i} (r \log_2(r) - (r - 1) \log_2(r - 1))^2 < m \sum_{i=1}^n \sum_{r=1}^{m_i} (\log_2(e) + \log_2(r))^2$$  \hspace{1cm} (3.19)

In (3.19), the inequality is obtained using the right hand inequality of (3.10). The inequality in (3.19) is expanded and simplified below using the inequalities (3.13, 3.14):

$$E(X^2) < m \sum_{i=1}^n \sum_{r=1}^{m_i} \left( (\log_2(e))^2 + 2 \log_2(e) \log_2(r) + (\log_2(r))^2 \right)$$
\[
\begin{align*}
\text{Var}(X) & < 3.1m^2 + m \sum_{i=1}^n \left[ m_i a_i^2 + 2m_i a_i + a_i^2 + 2a_i \right] - \left( \sum_{i=1}^n m_i a_i \right)^2 \\
& = 3.1m^2 + m \sum_{i=1}^n \left[ 2m_i a_i + a_i^2 + 2a_i \right] + \left\{ \left( \sum_{i=1}^n m_i \right) \sum_{i=1}^n m_i a_i^2 \right\} - \left( \sum_{i=1}^n m_i a_i \right)^2 \\
& = 3.1m^2 + m \sum_{i=1}^n \left[ 2m_i a_i + a_i^2 + 2a_i \right] + \sum_{i=1,i>j}^{n-1} \left[ m_i m_j (a_i - a_j)^2 \right] \quad (3.22)
\end{align*}
\]

In (3.22), the last step can be seen by careful expansion of the rightmost two bracketed quantities of the previous step and simplifying the resultant expansions systematically.

Using \( \log_2(m_i + 1) = \log_2[(m_i)(1 + 1/m_i)] \leq \log_2(m_i) + 1 \) for all \( m_i \geq 1 \), (3.20) can be written as

\[
E(X^2) < m^2 \left[ 1 + \log_2^2(e) \right] - mn
\]

\[
+ m \sum_{i=1}^n \left[ m_i \log_2^2(m_i) + 2m_i \log_2(m_i) + \log_2^2(m_i) + 2 \log_2(m_i) \right]
\]

\[
< 3.1m^2 + m \sum_{i=1}^n \left[ m_i \log_2^2(m_i) + 2m_i \log_2(m_i) + \log_2^2(m_i) + 2 \log_2(m_i) \right] \quad (3.21)
\]

The last step in (3.21) follows using the numerical value of \( \log_2(e) = 1.4427... \) and upper-bounding its square. Using \( E(X) = \sum_{i=1}^n m_i a_i, (a_i = \log_2(m_i)) \), the \( \text{Var}(X) \) in (3.18) can now be bounded as
Finally, for use in Chebyshev’s inequality [12], (3.22) is modified as:

\[
\frac{\text{Var}(X)}{E(X)^2} < \frac{3.1m^2 + m \sum_{i=1}^{n} [2m_i a_i + a_i^2 + 2a_i] + \sum_{i=1,j>i}^{n-1} [m_i m_j (a_i - a_j)^2]}{\left(\sum_{i=1}^{n} m_i a_i\right)^2} \\
= 3.1 \frac{m^2}{E(X)^2} + \left[ \frac{m}{E(X)} \right] \left[ \frac{2E(X) + \sum_{i=1}^{n} a_i^2 + 2a_i}{E(X)} \right] + \frac{\sum_{i=1,j>i}^{n-1} [m_i m_j (a_i - a_j)^2]}{E(X)^2}
\]

(3.23)

To proceed, assumptions are made on the value of \(E(X)\) as \(E(X) > m/\Delta\) with \(\Delta > 0\) in [13]. \(\Delta\) is taken as 1 in [14]. The entropy of \(n\) packet headers assuming uniform random distribution is \(H = \log_2(n)\). Entropy is related to the entropy norm [13] by \(H = (m \log_2(m) - E(X))/m\) where \(E(X) = F_H\) is the entropy norm. The expression for variance can now be bounded as

\[
\frac{\text{Var}(X)}{E(X)^2} < 3.1\Delta^2 + 3.53\Delta + O(\log_2(m))
\]

(3.24)

Except for the last term in (3.23) the rest of the expressions can be bounded by finite quantities. The last term must be of the form \(O(\log_2(m))\) i.e. \(k(\log_2(m))\) where \(k\) is conjectured to be \(< < 1\), based on the simulations conducted in Chapter 2. However, at the time of writing this thesis, it has not been possible to prove it mathematically. If this can be done, then the total number of counters can be reduced significantly which will be very effective for hardware implementation. This will be an area for future work.

While the procedure followed here in bounding the \(Var(X)\) has been done taking [14] as the starting point, what has been presented here is a completely different bounding technique and produces different results. Primarily, the key result in this section is the closed form expression in (3.23) which leads to (3.24). Though the authors of [14] claim that their bounds are the tightest in the literature, the authors in [13] have a much stronger theoretical bound. The same analysis used in this chapter to arrive at (3.23) can also be extended to the expression in [13].
CHAPTER 4
HARDWARE IMPLEMENTATION IN XILINX FPGA

The target Xilinx FPGA chosen for study of feasibility of the hardware implementation is the XC2VP30. The reason for this choice is that, the Xilinx University Program (XUP) board is available in the USF laboratory. The design is carried out using the Xilinx software tools version 9.2i. The software packages available in this version are the Xilinx ISE and the Xilinx EDK. Xilinx EDK 9.2 no longer supports the XUP board and the entire design was carried out using only the Xilinx ISE. Xilinx indicates that a future version of the EDK will support the board. The resources available in the XC2VP30 are: 27392 4-input LUTs, 136 18 kbit Block SelectRAM (bRAM, configurable in standard data widths only) and 136 $18 \times 18$ Block multipliers. There are upto 8 digital clock modules (DCMs) available on the chip which can be configured to increase the internal clock frequency. The period of the clock of the XUP board is 10 ns. Using these DCMs the chip can be internally clocked at a period of 5 ns. This is the maximum clock rate for the bRAM that was obtained during simulations for this chip. This is also the maximum performance estimate as given by the Xilinx bRAM notes [18]. ModelSim 6.1XE starter version is the tool of choice for post-route simulations of designs. The waveforms shown in this chapter are obtained from the post-route design simulated with ModelSim. The hardware description language in which design is carried out is Verilog.

The USF Academic Computing provided a minutes’s capture of IP data, that had approximately $2.6 \times 10^6$ packets. While it was indicated that this capture was based on a very low-traffic flow during off-peak hours, it is to be anticipated that normal IP data traffic will far exceed this number. An online free software [16] was used to extract this captured data and organize it into its constituent IP packet header features like source/destination.
MAC address, IP version, source/destination IP address, source/destination port, application using the IP protocol etc. A simple program was then developed to extract from this, only what is relevant for this work i.e. IP header source/address information. Only the IP destination addresses will be considered for all further analysis.

Normally on a 1 Gbit/sec router, based on an assumed average packet size of 300 bytes [3], approximately $450 \times 10^3$ number of packets can be expected in a second. On a 10 Gbit/sec router, this number would be approximately $4.5 \times 10^6$ in a second. The number of counters needed for the algorithm is based on the total number of packets $m$, fixed beforehand. As stated in Chapter 2, by trial and error, an epoch size of $m = 10^6$ was fixed, taking into account the resources on the FPGA board but without under-utilizing the board. Hence for all the hardware designs and simulations too, the epoch size of a million packets has been used.

4.1 Mapping the Algorithm into Hardware

The hardware design has to handle 32 bit data because in Internet Protocol, version 4 (IPv4), the source and destination addresses need 32 bits for storage. Although the newer IPv6 has larger address field of 128 bits, since the USF LAN data capture showed IPv4 headers, this version was used for the study. 32 bit registers are needed to store the packet destination addresses and the count value ($r$ in Code 2.1) for each of the randomly selected packets will be stored in 16 or 32 bit counters. Initially 16 bit register for the count value was selected to save hardware resources, subsequently in the bRAM based designs 32 bit count value is used. The 32 bit register needed to store the packet destination addresses serves a dual purpose: Initially it contains a random number that signifies when a packet will be picked for monitoring and secondly from that instant on it stores the address of the packet header that was available at that time instant on the bus. To be able to select the packet header (address) at a time instant, there is a need for global sample (sequence) count to be available to all modules so that, at the appropriate instant, the packet headers can be picked up. There is also the need for a global packet header bus on which the packet
addresses are placed in the design. The counter associated with each 32 bit packet header register, contains the count or in other words, the number of times that packet header address showed up in the data stream from its instant of sampling during the entire epoch. This is the basic module that implements the computation of \( r \), of the random variables \( X \) of Chapter 2 as shown in Code 2.1. The design requires \( s_1 \times s_2 \) such random variables, and hence that many counters. Once an epoch of \( m \) packet headers is analyzed for count values \( r \), then post-processing of the \( r \) to compute the logarithms and the value of the random variable \( X \) of step 3 of the Code 2.1 can be carried out. Finally the averaging and median computations of (2.3) of Chapter 2 are also required in the post-processing (post-epoch) phase. Lastly, in the pre-epoch phase, a random number generator to store initial values in the packet header register is also needed. If the epoch size is \( m = 2^{20} \) then a 20 bit random number generator would suffice from which the first \( s_1 \times s_2 \) values would be selected and placed into the packet header registers before an epoch begins. Thus the whole algorithm requires three phases, a pre-epoch phase (initialization phase), the epoch phase (monitoring phase) and a post-epoch phase (computation phase).

Ideally, high speeds can be achieved during an epoch, if \( s_1 \times s_2 \) (given by the expressions in Chapters 2, 3) packet header registers and counters which are used to monitor the data stream, can all operate in parallel. This is the basic counter module and is the first design considered. Only after exploring this module’s design space or modifications of it in the next two sections, is the post-processing phase considered in subsequent sections.

4.2 Design 1: Counter Module

The block diagram of the Counter Module is shown in Fig. 4.1

4.2.1 Design 1: Operation

The input to the module consists of two buses:

1. The 32 bit wide packet header bus (ph bus).
Figure 4.1. A Counter Module
2. A 16 or 20 bit wide (width depends on how many total packets \( m \), are planned apriori for the complete epoch) global sample or global sequence counter bus (gsc bus). For a million packets, this bus is 20 bit wide. Should the design call for monitoring more or less packets in an epoch, then this bus width can be increased or decreased to match the number of packets to be monitored. The contents of this bus, is the time instant for choosing packets whose sampling instant, a random number, is stored in the ph register during reset of this module. The value on the gsc bus is incremented every time a packet shows up on the packet header bus, and is synchronized with the positive edge of the clock.

A random number generator is needed for storing initial value in the ph register in the module. At the time of designing this module, one separate random number generator module for the entire design was envisaged which would be used to initialize the 32 bit ph registers with random values during reset. During testing of this design, the random number generator in Verilog was used in the testbench fixture. For testing, random numbers were also used to serve as packet headers. The ph bus served both, to initially carry the random numbers for sampling at reset into the ph register, as well as to carry the packet headers in the epoch phase. With reset high, the random number on the ph bus is loaded into the ph register by exercising the ld line of the register. Simultaneously, the D flipflop in the circuit is cleared. In testing since only random numbers are used for simulations, the ph bus always has a stream of random numbers with every clock cycle.

The two modes of operation in the Counter Module is as follows.

1. Reset/Load Phase: A high on the reset line makes the output of the D flipflop(f/f) low. The output of the f/f serves as the select signal \( S \) for the multiplexer. If \( S \) is low, the gsc bus is chosen for comparison, else the ph bus is chosen for comparison. When \( S = 0 \) the contents of the 32 bit register is compared with the gsc bus. If there is a match, the comparator has an output of 1 which then forces the output of the AND gate A high which serves as the load signal for the 32 bit register. The ph bus contents then gets stored in the register. Thus, the 32 bit register which
originally contained the random number which signals the time instant at which a
packet header (address) has to be picked, now stores the packet header available on
the ph bus and the flipflop is set to 1.

2. Increment Phase: When \( S = 1 \), the multiplexer chooses the packet header contents
for comparison. If there is a match with the 32 bit ph register containing the IP header
address to the contents of the ph bus, then both the f/f and the comparator output
are high which enables the other AND gate B to increment the 16 bit counter value
associated with the 32 bit ph register in the module.

A project with 16 such Counter Modules in parallel was simulated to test the design and
its post-route simulation is shown as in Fig. 4.2 and Fig. 4.3. Since now we need to address
each of the modules, there is additional circuitry needed for an address decoder and a
separate address enable (ae) line. During the initial phase of operation, when the reset and
the address enable (ae) signals are both high, random numbers available on the ph bus act
get stored into the 32 bit registers of every module. The address lines start at address 0 and
increment until address 15. During the reset phase the D flipflops in each module is also
cleared as explained above. When the reset goes low, the value on the gsc bus is checked
with the random number in the 32 bit register of each module, and if there is a match,
then the current contents of the ph bus, are loaded into the registers and the corresponding
counter value is set to 1. Thereafter, every time a new packet arrives on the ph bus, these
registers are checked for a match and the corresponding counters incremented.

4.2.2 Design 1: Advantages and Disadvantages

The Counter Module as described in the previous section is excellent for speed as
many such modules operating in parallel can achieve very high speeds only limited by
the maximum clock frequency possible on the XUP board and the DCMs. If the number of
modules \( = s_1 \times s_2 \), then all random variables \( X \) can be operated in parallel. By its design,
since it is a dedicated module, any number can be added, provided routing requirements
to multiple parallel modules are taken care of. However, The FPGA XC2VP30 has a total
Figure 4.2. Initial Set up for 16 Counter Modules Simulation Waveforms
Figure 4.3. Counters in Operation Simulation Waveforms
of 27392 4-input LUTs. The number of such Counter Modules that can be fitted on the board is only about 500 since each module occupied 54 LUTs. Since \( s_1 \times s_2 = 28525 \), even on the largest Virtex-II Pro chip in the Xilinx family, this design cannot be accommodated as only about 2000 Counter Modules can be fitted. The limitation of this design seems to be only with available resources and not with the design itself and hence, this design is abandoned. An alternative, although slower design, that could satisfy both the algorithm requirements as well as possibly be fittable on the XC2VP30 chip, is investigated in the next section.

4.3 Design 2: Counter Module Using the Block SelectRAM (bRAM)

4.3.1 Design Considerations

As described in the previous section, the parameters that we need to store and keep track of, are the IP destination address and the associated counter value. The aim is to be able to fit 28525 packet headers and \( (s_1 \times s_2 = 28525) \) counters. The IP packet header source/destination headers need 32 bit storage space. The count value (in the case of normal IP traffic patterns) for each of these distinct packet headers should normally require possibly only 16 bits to store a count value of not more than 65536. The usage of a bRAM fits the algorithm’s data structure of a matrix, because of its requirement of \( s_1 \times s_2 \) counters.

The Xilinx XUP Virtex-II Pro board has 136 bRAM each of which is 18 kbit [18]. The actual available number of blocks depend on the width of the bRAM used. If the data width is more than 32 bits wide, then the available number drops, because the Xilinx ISE through its IP core option cascades/tiles two or more blocks to accommodate the width needed. For example, since the IP addresses are 32 bits long, and if the counter values required is assumed 16 bits, then the memory width for the design would have been 48 bits. But when the design was simulated using 48 bits bRAM, the ISE ended up using 3 bRAMs instead of 1. The effective number of counters accommodated on the XC2VP30 then would a maximum of 11605 \( (\frac{136}{3} \times 256) \) counters instead of 28525 counters. Hence
this design has to be changed for something that would not end up using so many bRAMs, and at the same time, satisfy the counter requirements for an epoch of one million packets.

Configuring a 512 by 32 bRAM as a dual port RAM with 256 even address locations serving to hold packet header address and 256 corresponding odd address locations holding the counter value enables the use of just one bRAM per such configuration. Using 112 bRAMs and therefore 256 counters per bRAM in this configuration provides a total of 28672 counters and therefore allows an epoch of 1 million packets to be handled. Since both even and odd memory addresses can be simultaneously accessed and independently written to, this approach to storing the counters and packet address values is quite efficient. However the penalty is in speed. Since now, on each packet address available on the ph bus, every even location must be searched to see if the corresponding packet header exists at that location and if it exists then the corresponding counter at the odd address which is simultaneously available for incrementing is incremented and written back. To handle the initialization with random sampling instants at the even locations, the initial random numbers can be loaded through a file for the bRAM, that the ISE provides. However, this works only for simulations as hardware reset does not load the file again. The random number generator and entry of data into these bRAM locations will be addressed later for this design in Section 4.5. It is to be noted that for an epoch of a million packet headers, the maximum count value is not more than $2^{20}$ and therefore to code the counter value requires at most 20 bits per packet header. The extra bits available in the counters can be used to store additional information.

Thus by increasing the counter width to 32 bits instead of 16 bits, a better design emerges due to the limitations imposed by the Xilinx tools when using the bRAMs.

4.3.2 Design 2: Advantages and Disadvantages

This is a good design for fitting in several counters within one block. This hardware design also fits the general data structure (sketch) of the algorithm, i.e of the notion of having $s_1 \times s_2$ counters. Several such blocks, in this case, 16 bRAM blocks could be linked
Figure 4.4. Counter Module with bRAM Simulation Waveforms
to get 4096 counters > $s_1 = 4075$ (from Chapter 2) counters for an epoch of a million packets, and there would be $s_2 = 7$ such groups to cover the 112 bRAMs. Furthermore, the additional circuitry needed for comparing packet headers on the ph bus, global sample (sequence) counter values and for controlling the state machine of this module of 256 counters in one bRAM occupies 141 LUTs and allows the fittability of 112 such modules into the XC2VP30 with additional room to spare for other circuits. The drawback is now speed. A 5 ns clocking period (200MHz clocking frequency) of this design is possible. To compare a packet header with each location requires a READ state to read from memory the contents of the packet header area and counter value, a WAIT state to settle the outputs, a WRIT state when all comparisons and increments are made as appropriate and write back to memory locations of the contents, a LOOP state where address increment occurs or if all done a finish signal is raised to provide handshaking with the test bench module to provide the next packet header on the ph bus. The module goes back to a STRT state where it awaits a start signal from the test bench module when the next packet header is applied. Thus if $1 \leq N \leq 256$ counters are present in the bRAM, then the smallest time period before the next packet header can be applied is $1 + N \times 4$ clock periods. If $N$ is 256 then 1025 clock periods elapse before a new packet header can be processed by this design. Since all (112) bRAMs can be operated in parallel, with a 5 ns clocking period, the packet headers can be applied to the design at intervals of 5.125 $\mu$s. However it should be remembered that the design requires additional post-processing circuitry and the addition of these circuitry and post-route simulations of the composite design may change this maximum clocking period due to delays introduced by routing. Consequently a more realistic expectation for this design is a 10 ns clocking period leading to maximum packet header intervals of 10.25 $\mu$s. The simulations shown in Fig. 4.4 are with a 10 ns clocking period. The design allows trading off speed to number of counters in each bRAM. In the XC2VP100 with a total of 444 bRAMs, using 64 counters in each bRAM provides almost the required number of counters and the overall packet header throughput can be increased by 4 times from the numbers indicated above.
During the initial part of this work, it was planned that the Xilinx EDK 9.2i, would be used for simulating the above designs. The original design was to use the PowerPC on the XUP board to handle the post-processing phase of the algorithm. Since it was later realized that this version of EDK does not support the available Xilinx FPGA board, additional circuitry described in the next section for post-processing of the counter values had to be designed.

4.4 Logarithm Module

A logarithm (base 2) circuit (log circuit) is needed in the design to compute the random variables \( X_{ij} \) as shown in Code 2.1 with \( X = m \left( r \log_2(r) - (r - 1) \log_2(r - 1) \right) \). In Chapter 3 it was shown that this equation (3.10) is well approximated for large \( r \geq 4 \) to 5% error and for \( r \geq 15 \) to 1% error by the right or left hand side of (3.10). To keep the errors introduced by this approximation low, for values of \( r \leq 15 \), the equation in Code 2.1 is precomputed and looked up, while beyond that, it is computed using the right hand side of (3.10). The right hand side of (3.10) reduces the computation by having just 1 logarithm computation and one addition instead of 2 multiplications, 2 logarithms, 1 subtraction and 1 addition.

4.4.1 Design of the Log Circuit

The input to the log circuit is a positive integer and represents the final count value of how many times a packet showed up in the stream. Since the maximum epoch is assumed to handle upto a million packets, the maximum counter value is coded in 20 bits and the logarithm value therefore requires at most 5 bits for its integer portion. \( \log_2(2^{20}) = 20 \). If the overall logarithm value has to fit into an 8 bit register, then it leaves 3 bits for the fractional part. The algorithm of the log circuit is described in Code 4.1. The output of the log circuit has an overall accuracy of three places after the binary point. The maximum error is ±0.125. During normal operation, the final count value for each of the counters may be small, but during anomalies, they can reach high values, and one place
Begin
    MCnt = 0; SCnt = 0;
    Read InputNumber
    While (InputNumber ≠ 0)
        Begin
            Shift
            SCnt++;
        End
    Set Size
    Set base
    While ((base * Size) ≤ InputNumber)
        MCnt++;
    Return LogNum = SCnt + MCnt
End

Code 4.1. Pseudocode of Algorithm for Logarithm Computation

after decimal point is all that is required in such cases. So the choice of three places after binary point is quite adequate. The design uses the basic multiplier module available in the Xilinx library. There are three different ways a multiplier style can be configured on the Xilinx FPGA and these are as follows: 1. Block multiplier (preconfigured as an 18 × 18 bit multiplier producing 36 bit result. There are 136 such multipliers on the XC2VP30 chip). 2. LUT multiplier (bit size configured as required) and 3. Pipelined LUT multiplier (bit sizes configured as required but usually requires more LUTs than choice 2 for multiplier style). Of these three, preliminary simulations show that the pipelined LUT multiplier can be clocked the fastest, the LUT is the next fastest and the Block multiplier is slowest for this application. A choice of the Block multiplier would mean, that since the input is more than 18 bits, the Xilinx tools use 2 Block multipliers for each log circuit. A Block multiplier is efficient in terms of overall LUT utilization, because of dedicated hardware multiplier. LUT based multipliers need additional LUTs. With the LUT multiplier based design, the Logarithm Module takes 363 LUTs.
4.4.2 Log Circuit Performance

The log circuit takes a maximum of 29 clock cycles to compute logarithm of an integer from 1 to $2^{20}$ and is clockable at 5 ns period and this is the period used in the test results of this section. In Figure 4.5, the waveforms of the log circuit for computing the logarithm alone are shown. In Table 4.1 a comparison of the performance of the log circuit both in terms of numerical accuracy and speed of computation is made with Matlab computations of the log running on a 1.54 GHz Pentium (dual core) machine. The Matlab timing results in Table 4.1 are computed as an average of a million repeated logarithm computations. From the results of Table 4.1 it can be seen that the log circuit operates at a clock rate that is 8 times slower than the Pentium processor clock, but is still faster than Matlab computations. Only the logarithm computation timing is shown in the table as the lookup for count values $r \leq 15$ and addition needed in equation (3.10) can be accomplished in less than this time or in about the same time.
Table 4.1. Accuracy and Speed of the Log Circuit in Comparison with Matlab

<table>
<thead>
<tr>
<th>Number</th>
<th>Log of Number</th>
<th>Error</th>
<th>Time taken</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Log circuit</td>
<td>Matlab</td>
<td>200 MHz clk</td>
</tr>
<tr>
<td>47</td>
<td>5.625</td>
<td>5.5546</td>
<td>0.0704</td>
</tr>
<tr>
<td>243</td>
<td>7.875</td>
<td>7.9248</td>
<td>0.0498</td>
</tr>
<tr>
<td>999</td>
<td>10.0</td>
<td>9.9643</td>
<td>-0.0357</td>
</tr>
<tr>
<td>1024</td>
<td>10.0</td>
<td>10</td>
<td>0.0</td>
</tr>
<tr>
<td>7021</td>
<td>12.875</td>
<td>12.7775</td>
<td>0.0975</td>
</tr>
<tr>
<td>9999</td>
<td>13.375</td>
<td>13.2876</td>
<td>0.0874</td>
</tr>
<tr>
<td>25000</td>
<td>14.625</td>
<td>14.6096</td>
<td>-0.0154</td>
</tr>
<tr>
<td>50832</td>
<td>15.625</td>
<td>15.6334</td>
<td>0.0084</td>
</tr>
<tr>
<td>73222</td>
<td>16.25</td>
<td>16.1600</td>
<td>0.09</td>
</tr>
<tr>
<td>111111</td>
<td>16.75</td>
<td>16.7616</td>
<td>-0.0116</td>
</tr>
<tr>
<td>499502</td>
<td>18.875</td>
<td>18.9301</td>
<td>0.0551</td>
</tr>
<tr>
<td>637421</td>
<td>19.375</td>
<td>19.2819</td>
<td>0.0931</td>
</tr>
<tr>
<td>872119</td>
<td>19.75</td>
<td>19.7342</td>
<td>-0.0158</td>
</tr>
<tr>
<td>1000000</td>
<td>20.000</td>
<td>19.9316</td>
<td>-0.0684</td>
</tr>
</tbody>
</table>

4.5 RNG_Module

The RNG_Module is a random number generator. Xilinx supports loading the bRAM locations with initial values, using an external file. However, as stated in Section 4.3.1, this method works well only for simulations (one time). If the circuit is reset during a simulation or at the end of a simulation, the bRAM locations do not hold their original initial values. Hence a random number generator has to be part of the design along with additional circuitry needed to initialize the bRAM locations with random values on reset.

During the initialization phase, just before the beginning of an epoch, the bRAM (even) addresses are initialized with random numbers denoting the sampling instant. The random number generator design is a simple LFSR (linear feedback shift register) with feedback tap points well documented in the literature for various bit sizes [17]. This design is chosen because Matlab simulations of the histogram shown in Fig. 4.6 of the generated random numbers by this method shows a uniform distribution. The design selected uses a 20 bit LFSR and uses tap points [17] at various bit locations, with an initial seed that is provided in the 20 bit register at reset. Random numbers with a periodicity of 1048575 i.e \(2^{20} - 1\)
is generated by this random number generator and suffices for the million period epoch. A test run of the random number generator is shown in Fig. 4.7. This is sufficient for our applications as only 28672 values are needed at most (112 bRAMs × 256 counters in each bRAM). Using one such random number generator per module adds minimally to the overall LUTs (just one additional LUT) and as long as the seed provided in each module is relatively prime with the seeds in the other modules, the random number sequences for each bRAM module is independent. For test simulations the module with a smaller word size (8 bits) with appropriate taps [17] is used.
4.6 Integrated Module

The block diagram of the Integrated Module is shown in Fig. 4.8. To implement the whole algorithm for an epoch of $10^6$ packets, 112 such modules are needed. Due to design and routing constraints, it was necessary to design all three parts of the algorithm i.e., pre-epoch phase, the epoch phase, and the computing and local averaging of all the 256 random variables of post-epoch phase of the algorithm within the same module. Each Integrated Module contains a Counter Module using the bRAM, a RNG Module, and a Logarithm Module. Finally at the end of the epoch, it returns the average of all the 256
computed random variables i.e., computes $(\sum_{i=1}^{256} X_i)/256$. Simulation waveforms representing pre-epoch, epoch and post-epoch phases are shown in Fig. 4.9 through Fig. 4.13.

1. In Fig. 4.9 waveforms show the write signal wea begin enabled to write to each (even addresses) of the 256 locations of bRAM.

2. In Fig. 4.10 waveforms show bRAM location 12 has a random number = 3

3. In Fig. 4.11 waveforms show signals wea and web enabled to store packet header 9 at time instant 3. Here random numbers generated by the simulator act as packet headers.

4. In Fig. 4.12 waveforms show bRAM location 12 containing packet header 9 and the counter value i.e., bRAM location 13 incremented by 1.

5. In Fig. 4.13 waveforms show random variable $X$ computation of each counter value (odd addresses) and a moving sum of all 256 bRAM locations.
Figure 4.9. Pre-epoch Phase: Initialization
Figure 4.10. Epoch Phase: a
Figure 4.11. Epoch Phase: b
Figure 4.12. Epoch Phase: c
Figure 4.13. Post-epoch Phase: Random Variable Computation and Averaging
4.6.1 Hardware and Software Comparisons of the Integrated Module

A comparative summary of the software simulations with Matlab and hardware simulations in ModelSim of the Integrated Module using the same synthesized traffic is provided in this section in tabular form for the waveforms shown for the run of Fig. 4.9 through Fig. 4.13. The purpose is two fold: First, to provide a detailed comparison and verification of the working of the hardware, and second, to provide a comparison of the final estimates of the entropy norm provided by the hardware and software for a few runs.

The synthesized traffic is generated by using 1000 Verilog random numbers in the test bench for the Integrated Module. A 1000 packet post-route hardware simulation in ModelSim of one Integrated Module on a 1.54 GHz Pentium PC takes approximately 13 hours to run although this is only approximately 10.29 ms in actual hardware simulation time.

The traffic stream generated by the Verilog test bench and the random numbers generated by the hardware for the initial sampling instants stored at the 256 locations in bRAM were captured manually and used in the software (Matlab) simulations to verify the hardware simulations and check the correctness of the hardware design.

Just after the epoch phase is completed, the contents of the counters in bRAM and the corresponding counts from the software is depicted in Table 4.2 as \( r \). The values match exactly at all 256 locations in the Integrated Module. This proves the correctness of the epoch phase part of the module as well as the pre-epoch phase (initialization) aspects of the designed hardware module.

In the post-epoch phase, the random variable for each of these counters is computed as \( X/m = r \log_2(r) - (r - 1) \log_2(r - 1) \) in software (Matlab) in floating point and printed in Table 4.2 to 4 decimal places. In the hardware the corresponding \( X/m \) is computed using fixed point arithmetic as described earlier in this chapter and these are also shown in Table 4.2. The slight discrepancies are due to the use of fixed point arithmetic in hardware versus floating point arithmetic in software. The percentage error between these two are shown in the middle column of Table 4.2.
The Integrated Module in the post-epoch phase keeps a running total of the computed $X/m$ variables, concurrently. This cumulative sum from the hardware is shown progressively in Table 4.3 and the corresponding values obtained from floating point simulations in Matlab for the same run are also shown along with the error percentage between these two quantities.

Finally, the hardware computes the average of the random variables $X/m$ by right shifting by 8 bits (divide by 256) the cumulative total in the last clock cycle of the post epoch phase. The software computes the average by dividing by 256 in floating point. The results are 3.1875 and 3.2375 respectively, and represent $F_H/m$. The error between the two values is 1.5%. In Table 4.3 the final row shows the average value of the cumulative sum of all 256 random variables as computed by the hardware and software.

The entropy norm $F_H$ is $X/m(\text{avg}) \times 10^3$, where $10^3$ is the number of packets $m$, used in the hardware simulations shown in Fig. 4.9 through 4.13. Therefore, the final values of 3187.5 ($3.1875 \times 10^3$) from hardware and 3237.5 ($3.2375 \times 10^3$) from software are the estimates of the entropy norm ($F_H$) as computed with 256 counters (in 1 Integrated Module) and are within 1.5% of each other.

Two additional 1000 packet runs on two other different distributions were made, to show comparison between hardware and software estimates for $F_H$. The results are tabulated in Table 4.4. Each run of hardware simulations tabulated, represent approximately 10.5 ms of actual hardware time which includes the pre-epoch, epoch and post-epoch phases. Therefore, the post-epoch phase adds very little additional hardware computation time.

To follow the pattern of the data structure of the algorithm, 16 such modules ($256 \times 16 = 4096$) have to be grouped together, the average of all such modules, then becomes 1 of 7 ($s_2 = 7$) values. The median of 7 such averages multiplied by the epoch size, in this case, $m = 10^6$ is the value of the entropy norm. The block diagram of this approach is illustrated in Fig. 4.14. Each module has its own random number generator and should be provided with a different seed that is relatively prime to the others, to ensure that different modules do not pick up the same packets for monitoring purposes. Initialization is done, at the
Table 4.2. Comparison of Hardware Simulations Vs Software Simulations - Part 1

<table>
<thead>
<tr>
<th>Counter</th>
<th>Address</th>
<th>Hardware Simulations</th>
<th>Software Simulations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>$X/m$</td>
<td>$r$</td>
</tr>
<tr>
<td>1</td>
<td>02C</td>
<td>2.7500</td>
<td>0.1774</td>
</tr>
<tr>
<td>3</td>
<td>02C</td>
<td>2.7500</td>
<td>0.1774</td>
</tr>
<tr>
<td>5</td>
<td>034</td>
<td>3.2500</td>
<td>-0.1506</td>
</tr>
<tr>
<td>7</td>
<td>042</td>
<td>4.1250</td>
<td>0.4034</td>
</tr>
<tr>
<td>9</td>
<td>03A</td>
<td>3.6250</td>
<td>-0.4255</td>
</tr>
<tr>
<td>11</td>
<td>03E</td>
<td>3.8750</td>
<td>0.6445</td>
</tr>
<tr>
<td>13</td>
<td>020</td>
<td>2.0000</td>
<td>0</td>
</tr>
<tr>
<td>15</td>
<td>034</td>
<td>3.2500</td>
<td>-0.1506</td>
</tr>
<tr>
<td>17</td>
<td>034</td>
<td>3.2500</td>
<td>-0.1506</td>
</tr>
<tr>
<td>19</td>
<td>034</td>
<td>3.2500</td>
<td>-0.1506</td>
</tr>
<tr>
<td>21</td>
<td>020</td>
<td>2.0000</td>
<td>0</td>
</tr>
<tr>
<td>23</td>
<td>02C</td>
<td>2.7500</td>
<td>0.1774</td>
</tr>
<tr>
<td>25</td>
<td>03E</td>
<td>3.8750</td>
<td>0.6445</td>
</tr>
<tr>
<td>27</td>
<td>03E</td>
<td>3.8750</td>
<td>0.6445</td>
</tr>
<tr>
<td>29</td>
<td>042</td>
<td>4.1250</td>
<td>0.4034</td>
</tr>
<tr>
<td>483</td>
<td>03A</td>
<td>3.6250</td>
<td>-0.4255</td>
</tr>
<tr>
<td>485</td>
<td>03E</td>
<td>3.8750</td>
<td>0.6445</td>
</tr>
<tr>
<td>487</td>
<td>034</td>
<td>3.2500</td>
<td>-0.1506</td>
</tr>
<tr>
<td>489</td>
<td>02C</td>
<td>2.7500</td>
<td>0.1774</td>
</tr>
<tr>
<td>491</td>
<td>03E</td>
<td>3.8750</td>
<td>0.6445</td>
</tr>
<tr>
<td>493</td>
<td>046</td>
<td>4.3750</td>
<td>-0.6090</td>
</tr>
<tr>
<td>495</td>
<td>03E</td>
<td>3.8750</td>
<td>0.6445</td>
</tr>
<tr>
<td>497</td>
<td>02C</td>
<td>2.7500</td>
<td>0.1774</td>
</tr>
<tr>
<td>499</td>
<td>034</td>
<td>3.2500</td>
<td>-0.1506</td>
</tr>
<tr>
<td>501</td>
<td>03E</td>
<td>3.8750</td>
<td>0.6445</td>
</tr>
<tr>
<td>503</td>
<td>020</td>
<td>2.0000</td>
<td>0</td>
</tr>
<tr>
<td>505</td>
<td>034</td>
<td>3.2500</td>
<td>-0.1506</td>
</tr>
<tr>
<td>507</td>
<td>02C</td>
<td>2.7500</td>
<td>0.1774</td>
</tr>
<tr>
<td>509</td>
<td>03A</td>
<td>3.6250</td>
<td>-0.4255</td>
</tr>
<tr>
<td>511</td>
<td>02C</td>
<td>2.7500</td>
<td>0.1774</td>
</tr>
<tr>
<td>Counter Addr</td>
<td>Hardware Simulations</td>
<td></td>
<td>Software Simulations</td>
</tr>
<tr>
<td>-------------</td>
<td>----------------------</td>
<td>---</td>
<td>----------------------</td>
</tr>
<tr>
<td></td>
<td>Cum Sum of X/m</td>
<td>% Error</td>
<td>Cum Sum of X/m</td>
</tr>
<tr>
<td><strong>Addr</strong></td>
<td><strong>Hex</strong></td>
<td><strong>Decimal</strong></td>
<td><strong>Hex</strong></td>
</tr>
<tr>
<td>1</td>
<td>0002C</td>
<td>2.7500</td>
<td>0.1774</td>
</tr>
<tr>
<td>3</td>
<td>00058</td>
<td>5.5000</td>
<td>0.1774</td>
</tr>
<tr>
<td>5</td>
<td>0008C</td>
<td>8.7500</td>
<td>0.0558</td>
</tr>
<tr>
<td>7</td>
<td>000CE</td>
<td>12.8750</td>
<td>0.1675</td>
</tr>
<tr>
<td>9</td>
<td>0010E</td>
<td>16.5000</td>
<td>0.0378</td>
</tr>
<tr>
<td>11</td>
<td>00146</td>
<td>20.3750</td>
<td>0.1537</td>
</tr>
<tr>
<td>13</td>
<td>00166</td>
<td>22.3750</td>
<td>0.1400</td>
</tr>
<tr>
<td>15</td>
<td>0019A</td>
<td>25.6250</td>
<td>0.1032</td>
</tr>
<tr>
<td>17</td>
<td>001CE</td>
<td>28.8750</td>
<td>0.0747</td>
</tr>
<tr>
<td>19</td>
<td>00202</td>
<td>32.1250</td>
<td>0.0520</td>
</tr>
<tr>
<td>21</td>
<td>00222</td>
<td>34.1250</td>
<td>0.0489</td>
</tr>
<tr>
<td>23</td>
<td>0024E</td>
<td>36.8750</td>
<td>0.0585</td>
</tr>
<tr>
<td>25</td>
<td>0028C</td>
<td>40.7500</td>
<td>0.1145</td>
</tr>
<tr>
<td>27</td>
<td>002CA</td>
<td>44.6250</td>
<td>0.1608</td>
</tr>
<tr>
<td>29</td>
<td>0030C</td>
<td>48.7500</td>
<td>0.1814</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>483</td>
<td>030E0</td>
<td>782.0000</td>
<td>0.0610</td>
</tr>
<tr>
<td>485</td>
<td>0311E</td>
<td>785.8750</td>
<td>0.0639</td>
</tr>
<tr>
<td>487</td>
<td>03152</td>
<td>789.1250</td>
<td>0.0630</td>
</tr>
<tr>
<td>489</td>
<td>0317E</td>
<td>791.8750</td>
<td>0.0634</td>
</tr>
<tr>
<td>491</td>
<td>031BC</td>
<td>795.7500</td>
<td>0.0663</td>
</tr>
<tr>
<td>493</td>
<td>03202</td>
<td>800.1250</td>
<td>0.0626</td>
</tr>
<tr>
<td>495</td>
<td>03240</td>
<td>804.0000</td>
<td>0.0654</td>
</tr>
<tr>
<td>497</td>
<td>0326C</td>
<td>806.7500</td>
<td>0.0658</td>
</tr>
<tr>
<td>499</td>
<td>032A0</td>
<td>810.0000</td>
<td>0.0649</td>
</tr>
<tr>
<td>501</td>
<td>032DE</td>
<td>813.8750</td>
<td>0.0677</td>
</tr>
<tr>
<td>503</td>
<td>032FE</td>
<td>815.8750</td>
<td>0.0676</td>
</tr>
<tr>
<td>505</td>
<td>03332</td>
<td>819.1250</td>
<td>0.0667</td>
</tr>
<tr>
<td>507</td>
<td>0335E</td>
<td>821.8750</td>
<td>0.0671</td>
</tr>
<tr>
<td>509</td>
<td>03398</td>
<td>825.5000</td>
<td>0.0649</td>
</tr>
<tr>
<td>511</td>
<td>033C4</td>
<td>828.2500</td>
<td>0.0653</td>
</tr>
</tbody>
</table>

\[
X/m(\text{avg}) = 3.1875(033H) \quad 1.5444 \quad X/m(\text{avg}) = 3.2375
\]
Table 4.4. $F_H$ Computation Comparison - One Integrated Module

<table>
<thead>
<tr>
<th>Packet Header Range</th>
<th>Hardware Simulations $F_H = \bar{X}/m(\text{avg}) \times 10^3$</th>
<th>% Error</th>
<th>Software Simulations $F_H = \bar{X}/m(\text{avg}) \times 10^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 ... 255</td>
<td>$3.1875 \times 10^3$</td>
<td>1.5444</td>
<td>$3.2375 \times 10^3$</td>
</tr>
<tr>
<td>1 ... 49</td>
<td>$5.6875 \times 10^3$</td>
<td>-1.9065</td>
<td>$5.5811 \times 10^3$</td>
</tr>
<tr>
<td>0 ... 10</td>
<td>$7.8125 \times 10^3$</td>
<td>-0.7298</td>
<td>$7.7559 \times 10^3$</td>
</tr>
</tbody>
</table>

time of the reset signal, before the epoch starts. This integrated design approach is highly modular, self contained and easily scalable.

Approximately 585 LUTs are needed per module and the XC2VP30 can now only accommodate overall $27392/585 \approx 46$ such modules. Thus in either case, using single counter modules (of Section 4.2) or the bRAM based Counter Module (of Section 4.3) the XC2VP30 cannot hold the entire design.

An alternative design approach would be to have the Counter Module with bRAM as one separate module, and have either one or several (optional) Logarithm Modules as the other module for post-epoch processing. This design would probably fit comfortably in the XC2VP30 FPGA, and the number of LUTs used overall would be reduced. But at the time of connecting the modules together, it was realized that Xilinx does not support enough tri-state buffers to isolate bRAMs from a common bus. A multiplexed design approach would consume additional LUTs to connect each of the 112 modules one by one to the Logarithm Module along with additional circuitry for addressing each of these modules. This design approach would greatly reduce the speed of the post-epoch phase. Hence it had to be modified to something that would access the bRAM output bus, without bus contention. The cost to this is embedding the Counter Module with bRAM into the Logarithm Module. This would take additional LUTs not only because the additional module would be embedded into each block, but also because of the additional routing and states needed to control the Integrated Module.

While 112 of the Integrated Modules do not fit on the XC2VP30, the XC2VP100 chip in the same Virtex-II Pro family of with resources of 444 bRAMs, 444 Block multipliers
and $99 \times 10^3$ LUTs can accommodate the entire design quite comfortably as envisaged. Other FPGA boards in the same Virtex-II Pro family handle this requirement.
Figure 4.14. Block Diagram of all 112 Modules and their Interconnect
The estimate of the entropy norm as discussed in Chapter 2, is the median of all $s_2 = 7$ values. A hardware module based on a simple sorting algorithm, the selection sort, is used to sort the input numbers and pick the median. The input to the module is a set of values, each of which, represents the average of the random variables of 16 Integrated Modules as shown in Fig. 4.14, computed during the post-epoch phase. The hardware simulation waveforms are shown in Fig. 4.15 and this module occupies 309 LUTs.

Figure 4.15. Median Module Simulation Waveforms

### 4.7 Median Module

The estimate of the entropy norm as discussed in Chapter 2, is the median of all $s_2 = 7$ values. A hardware module based on a simple sorting algorithm, the selection sort, is used to sort the input numbers and pick the median. The input to the module is a set of values, each of which, represents the average of the random variables of 16 Integrated Modules as shown in Fig. 4.14, computed during the post-epoch phase. The hardware simulation waveforms are shown in Fig. 4.15 and this module occupies 309 LUTs.
CHAPTER 5
CONCLUSIONS

The current work has concentrated on two aspects: first of all to find the possibility of reducing the number of overall counters, and secondly to study the feasibility of hardware (FPGA) implementation of the randomized algorithm in [13] for entropy norm computation of IP data streams.

In Chapter 2, some theoretical aspects as well as Matlab simulations of the algorithm were discussed. Simulations were carried out for both a uniformly random distribution of packet headers, and skewed distribution. This initial simulation study was necessary to study the overall algorithm structure as well as to get a feel for the total number of counters needed for the algorithm.

In Chapter 3, it was shown that it is possible to reduce the overall number of counters by about 30% just by reducing the $s_2$ counters. This is necessary to fit a million packet epoch conceptually on the XC2VP30 FPGA. Further tightening of the variance bound was attempted in Chapter 3, and a new closed form expression for $\frac{\text{Var}(X)}{E[X]^2}$ has been derived. Although it has been conjectured that the constant $k << 1$ for the last term in expression (3.24), at the time of writing this thesis, it has not been possible to prove it mathematically, and give a much stronger bound on the overall expression for $\frac{\text{Var}(X)}{E[X]^2}$. This area will be a subject for future work. A tighter bound in the closed form expression as shown in (3.24) will significantly reduce the number of counters overall, and hence would be much more feasible for a possible ASIC/FPGA implementation.

In Chapter 4 hardware designs along with their simulations were presented. A design approach has been explored with design trade-offs. Constant modifications had to be made in order to balance routing to all modules, while at the same time come up with a
design that could fit into the hardware and operate at speeds capable of real-time packet handling. The averaging of the logarithm computations of all the 256 counters available in 1 bRAM and median computations can be accomplished after the epoch is completed and the entropy norm computation is achievable in near real-time in a FPGA. Table 5.1 summarizes the total number of LUTs each designed module occupies.

The hardware designs discussed in Chapter 4 can broadly be classified as supporting three different phases of operation of the algorithm.

1. Pre-epoch phase
   In this phase, the registers that ultimately store the packet headers must be initialized with random numbers generated by the RNG Module (Section 4.5). If the Integrated Module design approach is considered, since this will be replicated with every module, initial seeds to each of these modules have to be such that the same random sequence is not replicated. This is easily achieved by choosing numbers that are relatively prime.

2. Epoch phase
   Counter Module (Section 4.2) and Counter Module with bRAM (Section 4.3) discussed two design options. In this phase, packet headers will be stored into registers based on the time instant in the registers, and monitoring of the packets will be done. Counters associated with each packet header increments every time a packet header once sampled repeats in the stream. The epoch for this study was fixed at 1 million packets.
   The first design is excellent for high processing speeds. Row 1 of Table 5.2 shows packet handling capability of this design. The second design is a trade off between speed and device utilization. The bRAM locations representing the counters contain the frequency of packet headers for that epoch.

3. Post-epoch phase
   Logarithm Module (Section 4.4) and Median Module (Section 4.7) are needed to
Table 5.1. Device Utilization Summary of FPGA Synthesis on XC2VP30

<table>
<thead>
<tr>
<th>Design type</th>
<th>bRAM block</th>
<th>Logic Utilization</th>
<th>Route through</th>
<th>No of 4 input LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counter Module(^a)</td>
<td>None</td>
<td>37</td>
<td>46</td>
<td>83</td>
</tr>
<tr>
<td>Counter Module(^b)</td>
<td>None</td>
<td>593</td>
<td>271</td>
<td>864</td>
</tr>
<tr>
<td>Counter Module with bRAM</td>
<td>1</td>
<td>85</td>
<td>62</td>
<td>147</td>
</tr>
<tr>
<td>Logarithm Module</td>
<td>None</td>
<td>343</td>
<td>20</td>
<td>363</td>
</tr>
<tr>
<td>Median Module</td>
<td>None</td>
<td>293</td>
<td>16</td>
<td>309</td>
</tr>
<tr>
<td>RNG Module</td>
<td>None</td>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>Integrated Module</td>
<td>1</td>
<td>540</td>
<td>44</td>
<td>584</td>
</tr>
</tbody>
</table>

\(^a\)1 Module Simulated  
\(^b\)16 Modules Simulated

compute the entropy norm of the IP data stream. In this phase, the random variables associated with each counter of the bRAM will be computed using the Logarithm Module and average of all locations of 1 bRAM will be computed. Several such modules have to be grouped, and further, their average computed. The median of several \(s_2\) such averages will give \((F_H/m)\).

The final design, Integrated Module (Section 4.6) presented is highly modular, self-contained and scalable. For large scale applications the existing hardware can be scaled up by adding Integrated Modules. Should the design call for entropy norm estimation of other packet features, this information can be loaded on the bus, instead of the destination address that is presently used. Should simultaneous entropy norm calculations be desired for multiple features such as destination addresses and port numbers then duplication of the modules and buses will be needed.

5.1 Alternative Design Considerations

In Chapter 4, the Counter Module using the bRAM was discussed. The design considered had 256 bRAM locations for counters, i.e., fully utilized each bRAM available, for optimality. This leads to slower speed of operation while providing better utilization of the FPGA. Higher processing speeds can be achieved if the design considered, utilized only 32 bRAM locations for counters. In Table 5.2 row 5, the increase in the packet handling
capacity with this approach is shown. But then, more bRAMs would be needed and an increase in routing should also be taken into account.

## 5.2 Future Work

The next goal would be to find how to interface with the designed hardware and to study the performance of the resultant hardware on IP data traffic.

The current study is by no means complete and exhaustive in terms of being able to detect anomalies. Due to the challenging nature of the problem area, sustained and detailed statistical studies have to be made using entropy results of not just one packet feature but simultaneously several different features to arrive at conclusions. This takes considerable amount of effort. Studies have to made at different times of the days over several months to study the traffic pattern and then use this information to draw some conclusions about anomalous behavior of the stream. The challenges that lie ahead would be in the following issues:

1. What threshold to consider, for the entropy norm, in order to set up a conclusive boundary between normal and anomalous traffic behavior.

2. What is a good epoch period for monitoring such traffic patterns.

Future study should also focus on reducing the number of counters because a vastly reduced number of counters will be more conducive to an ASIC/FPGA implementation. On the
hardware side, focus should be on interfacing the data from network links to provide appropriate input to the existing hardware for entropy norm computations.
REFERENCES


[4] Personal communications with Dr. R. Tripathi, University of South Florida, Tampa.


