FPGA based Network Traffic Analysis using Traffic Dispersion Patterns

Faisal Khan
Lawrence Livermore National Labs
fnkhan@ucdavis.edu

Maya Gokhale
Lawrence Livermore National Labs
maya@llnl.gov

Chen-Nee Chuah
University of California, Davis
chuah@ucdavis.edu

***Abstract*—The problem of Network Traffic Classification (NTC) has attracted significant amount of interest in the research community, offering a wide range of solutions at various levels. The core challenge is in addressing high amounts of traffic diversity found in today's networks. The problem becomes more challenging if a quick detection is required as in the case of identifying malicious network behavior or new applications like peer-to-peer traffic that have potential to quickly throttle the network bandwidth or cause significant damage. Recently, Traffic Dispersion Graphs (TDGs) have been introduced as a viable candidate for NTC. The TDGs work by forming a network wide communication graphs that embed characteristic patterns of underlying network applications. However, these patterns need to be quickly evaluated for mounting real-time response against them. This paper addresses these concerns and presents a novel solution for real-time analysis of Traffic Dispersion Metrics (TDMs) in the TDGs. We evaluate the dispersion metrics of interest and present a dedicated solution on an FPGA for their analysis. We also present analytical measures and empirically evaluate operating effectiveness of our design. The mapped design on Virtex-5 device can process 7.4 million packets/second for a TDG comprising of $10k$ flows at very high accuracies of over 96%.**

I. Introduction

Network Traffic Classification (NTC) is keystone in a wide range of network applications such as detection of anomalies and security attacks, identifying new applications, and traffic engineering. A number of critical network management decisions such as blocking or re-routing application traffic, or detection of anomalies, require real-time network traffic analysis. A high-quality network measurement tool is crucial in judiciously making such decisions.

The authors thank John May at LLNL and Soshant Bali at UC-Davis for their useful feedback and suggestions in the course of this work.

Auspices: This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. IM release number (LLNL-CONF-426380)

Recent work on NTC algorithms and techniques has yielded a flurry of proposed approaches. The methods can be grouped by their level of observation: (a) packet level, such as traditionally used TCP port numbers [1] or packet inspection for application signatures [2], (b) flow level statistical techniques that classify flows based on flow duration, number and size of packets per flow, and inter-packet arrival time [3], [4], and (c) host level, such as host-behavior approaches [5], that form a social interaction pattern of hosts before classifying their traffic.

The above techniques fall-in at various levels of accuracy and feasibility. The packet level techniques, though being more accurate, are increasingly facing difficulties with applications camouflaging their communication, for instance by employing encryption. They also involve significant privacy and legal concerns. The aggregation of packets into flows and grouping of flows over a host offer further insights but at the cost of increase in computational requirements as well as classification latency.

The aggregation of network wide host communication patterns in a "who talks to whom" *Traffic Dispersion Graph (TDG)* has been recently proposed as a natural extension in NTC schemes [6]. The 'who' and 'whom' in the graph are the communicating hosts, or graph nodes, while the edges between them define a type of communication that takes place between them. The TDGs have been employed for identifying network intrusion [6], worm propagation [7] and more recently for application identification, such as peer-to-peer (P2P) traffic [8], [9]. Depending on the requirements, such as quickly identifying worm, the graph analysis needs to be fast enough to limit the worm effects and mount a response. However, little attention has been given to the details of a real-time TDG implementation and analysis.

Recently, FPGAs have been seen to support real-time network traffic measurements [10]. However a real-time and accurate maintenance of a complicated data structure such as a graph is not a trivial problem, especially in hardware, as it poses enormous computational and resource requirements. They further require visual interpretations that may involve human intervention. Instead, if the graph can be summarized using graph-metrics such as amount of connectivity, size etc, it can lead to a quick evaluation of the TDGs. Such graph metrics, which we call Traffic Dispersion Metrics (TDMs), have also been the focus of previous studies.

TDGs have traditionally been processed offline, where the analysis has a full view of the dispersion graph. A real-time analysis of a TDG requires a complete rethink on what needs to be collected in the TDG and its associated cost/benefit analysis, specifically, the accuracy and the resource budget trade offs. In this paper, we provide some answers to these questions and present a hardware implementation supporting real-time evaluation of TDGs. We provide analytical and empirical measurements for determining accuracy and throughput that define the operating effectiveness of our solution. We have mapped our solution to Xilinx Virtex-5 device where it is seen to offer superior accuracy of over 96% having a throughput of 7.9 million packets/second while utilizing just 10% of logic and 73% of device RAM resources.

II. Background

A Traffic Dispersion Graph (TDG) is a directed graphical representation of various communications occurring among a set of entities. These entities in an IP network could be hosts with distinct IP addresses forming nodes in the graph while the edges represent interaction patterns among them. Thus one of the core question in the TDGs is what defines the edge, or an edge-filter. The fundamental question can be answered while taking into account the context of the study. An edge could be establishment of a connection, like TCP, between two entities, exchange of certain number of packets or bytes, a specific pattern in the packet payload or communication over a port or set of ports. These studies can result in different types of applications like identifying a new application over a port or spread of a worm with a specific signature.

We hereby present a design that is self-contained in that it does not need to maintain a TDG, but rather TDMs are maintained in real-time. The idea is to identify a set of TDMs that are easy to compute in hardware while at the same time representative enough to characterize the graph in a passing data stream. We make use of some of the metrics discussed in [6].

TDMs can be classified as online or offline. The online metrics can evaluated in passing data-streams whereas offline metrics need full view of a graph for their accurate evaluation. Some of the metrics are:

1) Degree: The degree of a node, or a unique address, in the graph represents its number of edges. Since TDGs are directed, an InDegree represent number of incoming edges, or connections, while an OutDegree represents number of outgoing edges, or connections that the node initiates. Some associated measures include average and maximum degree in the TDG. A high InDegree represents popularity of a host while a high OutDegree may represent diversity of communication of a client.

2) In-&-Out: The In-&-Out, or InO, measure represents the number of nodes that have both incoming and outgoing edges, or in other words, they act both as initiator and destination of communication in a TDG. As we will see, the InO measure has a strong correlation with P2P type applications.

3) Giant Connected Components (GCCs): The GCC measure defines the percentage of nodes that are connected together in the graph, or the sizes of latent sub-graphs within a TDG.

4) Depth: The depth of a node in the TDG defines the length of a sequence of nodes that are communicating with each other. The measure is therefore useful in quantifying the spread of communication. For instance, a high average depth in a TDG may represent contagious nature of some application, typically a worm.

The Degree, In-&-Out and Density in the above are online TDMs while the rest require an offline evaluation.

A. Bloom Filters

Bloom filters are space-efficient probabilistic data structures that are used to test if an element (key), X, is a member of a set. It works by having an array of size m bins that are indexed using k hash functions, each generating a hash value. To program the element X in the set, k bits corresponding to the k hash values are set in the array. The membership query process works in reverse by checking to see if all the k values corresponding to an element were set. If at least one of the k values is not found to be set, then the element is declared to be a non-member of the set. However, if all the values are seen to be set, the membership is declared true with certain probability. This false positive probability comes from the fact that the k values can be set by any of the already programmed n elements. Thus a Bloom-filter have a zero false negative (f_n) but a limited false positive (f_p) probability. This false probability is expressed as

$$f_p = [1 - (1 - \frac{1}{m})^{nk}]^k \approx (1 - e^{-nk/m})^k \tag{1}$$

It can be seen that the rate of false positives in a Bloom-filter can be reduced by having appropriate values for k and m for a given n. An optimal false positive probability yields the following relation between k, m and n

$$k = (m/n)ln2 \tag{2}$$

that actually corresponds to

$$f_p = (1/2)^k \tag{3}$$

Similar to false positive, one can also derive true negative (t_n) probability of the Bloom-filter as

$$t_n = (1 - \frac{1}{m})^{nk^2} \approx e^{-nk^2/m} \tag{4}$$

that gives us the probability of a ‘Yes’ answer from the Bloom-filter (true and false positives) as

$$p_y = t_p + f_p = 1 - t_n = 1 - e^{-nk^2/m} \tag{5}$$

Bloom filters are extended to support counting operation by extending individual bin sizes from a single bit to a p-bit counter. Such Counting Bloom Filters (CBFs) can record number of times an element X was programmed into the filter.

Name	Date/Time	Type	Duration	Unique Nodes	5-tuple Flows	Avg. Utilization Mbps
WIDE	2006-03-03/13:00	Backbone	2h	1,041,622	4,670,259	31.0(9.7)
OC48	2006-01-15/10:00	Backbone	1’02h	2,945,800	22,109,681	589.0(127.8)

TABLE I
THE SET OF PUBLICLY AVAILABLE TRACES USED

Rule	Description
A	packets > 0
B	packets > 1
C	packets > 1 & avg pkt-size > 60

TABLE II
RULE EXPLANATION

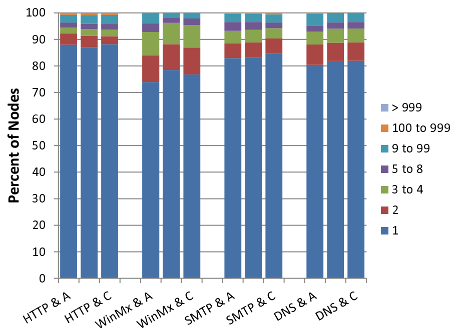

Fig. 1. InDegree Distribution

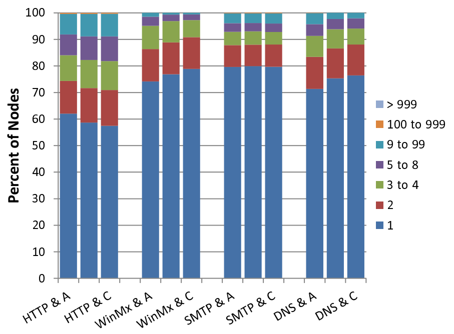

Fig. 2. OutDegree Distribution

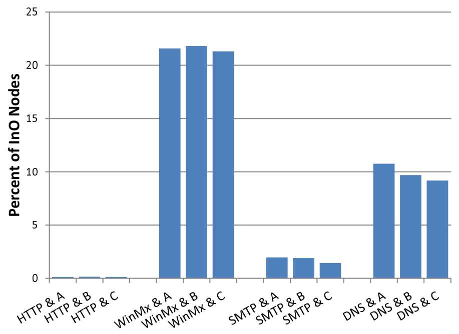

Fig. 3. InO Distribution

The insert operation then involves incrementing the k counters instead of setting them up. The presence of an element X in CBF then involves checking that all the k hash counters associated with the element have non-zero count values.

III. QUANTIFYING TDGS

The focus of this work is in real-time quantification of TDGs. We therefore focus on TDMs that can be evaluated online such as Degree and InO measures. We use a port based edge filter that goes along with the lines of the study, i.e, to detect the type of an application on a given port. However, as opposed to previous studies where the port based filter is enriched with a three way TCP handshake [6], we experiment with simpler edge filters. The three-way handshake occurs end-to-end at the application-layer in an OSI model and is quite resource expensive to be detected at our lower network-layer. We suspect that simpler filters can yield equally efficient classification rules while reducing hardware resource budgets.

We use a number of ports belonging to legacy applications, such as port-80 for HTTP, port-25 for SMTP, port-53 for DNS, and ports-$6699/6257$ for a P2P WinMX application. We used the data-traces shown in Table-I in our experiments. The port based edge filters were next enriched using simple rules such as edge on first packet, edge if there is more than 1 packet, and edge if there are more than a packet and the packet sizes are greater than 60 bytes, shown in Table-II. The rationale behind them is to incorporate the handshake information, without explicitly checking for the handshake. For instance, an edge filter involving packet size being at least 60 bytes filters out flow-control packets involving only headers, thereby making it more likely that a valid connection in the direction of data-flow gets logged. Similarly, ensuring multiple packets have been seen in a given direction further strengthens that a valid connection has been established.

The InDegree, OutDegree and InO plots so obtained are shown in Figures 1-3. It can be noted that the more enriched edge-filters tend to reduce the noise within the legacy applications, as is evident with decreasing InO percentages of DNS and SMTP. However, the general trends of the applications, as first reported in [6], are indeed present in the TDMs. The InO stands out as a strong indicator for presence of P2P based activity. DNS, with its moderately high InO, comes as a potential candidate to be misclassified as P2P application. However, its characterisitic behavior involves a very few supernodes with very high InDegrees and OutDegrees, as represented with a very slim bar at the top ($100 - 999$ bin). In contrast, the P2P supernodes have significantly smaller degrees (less than 99). This difference in signature was used in [8] where the authors proposed using a medium average degree of 2.8 along with high InO to isolate P2P activity.

However, calculating moving average is not trivial in hardware and requires significant logic. We therefore slightly modify the above rule for classifying P2P activity as: if 20% of InO nodes are present and there are no nodes in the high degree bins, then a P2P activity can be declared at a given port. Given these observations, we now turn our attention on designing an architecture that can help evaluate the TDMs in real-time.

IV. SYSTEM DESCRIPTION

A high level diagram of the system is presented in Figure-4. The design employs two Bloom filters, the flow-filter and Input-Output filter (InO) and two CBFs, the Source CBF (SC) and the Destination CBF (DC). There is also a packet-filter that performs user-defined edge filtering on incoming packets. The filtered two-tuple $\{\text{source } (s_i), \text{destination } (d_i)\}$ flows are forwarded to the controller that uses them at the various filters and counters to update the statistics.

As discussed earlier, the Bloom Filters (CBFs) efficiently answer queries like if (or how many times) a key was programmed. It is however not trivial to use the them to answer queries like *'how many of the unique keys have been programmed?'* or *'how many keys in the counter have a count equal to a certain value?'*. Our solution to answer such questions is by maintaining a statistic unit that explicitly stores answers to these questions. The unit is updated whenever updates are being made in the filters.

The Statistic Unit (SU) stores scalar as well as vector quantities. The scalar quantities include the total number of unique addresses (or nodes) and InO nodes that have been seen so far. The vector quantities include InDegree and OutDegree distributions. The bin sizes for the distributions are however logarithmic to the base of 2, which simplifies their addressing mechanism in hardware by using simple shift operations.

The part of control algorithm for checking and updating values in statistics unit due to source-addresses is next presented. A parallel executing algorithm utilizing destination-address at the controller is omitted for brevity.

```
1: for $i = 1$ to $n$ do
2:   if $(s_i, d_i) \notin CF$ then
3:     $CF \leftarrow (s_i, d_i)$
4:     if $s_i \notin SC$ then
5:       $nodes \leftarrow nodes + 1$
6:     end if
7:     $SC\{s_i\} \leftarrow SC\{s_i\} + 1$
8:     addr $\leftarrow log_2\{SC\{s_i\}\}$
9:     $OutDegree\{addr\} \leftarrow OutDegree\{addr\} + 1$
10:    if $s_i \in DC$ and $s_i \notin InO$ then
11:      $InO \leftarrow s_i$
12:      $InO\_nodes \leftarrow InO\_nodes + 1$
13:    end if
14:  end if
15: end for
```

The controller first checks the incoming flow (s_i, d_i) in the flow-filter. A successful match in the filter implies a previously seen edge which is not processed further. However, for a new flow, its individual addresses are used at CBFs to update the degrees. A special case of zero degree also implies a new address which is used for node-count update. Finally, the incoming addresses are considered for being InO nodes by checking the respective addresses in opposite CBFs, i.e. Source address is DC and destination-address in SC. A true (non-zero degree) in either case signals an InO node. The originality of the InO node is checked in the InO filter, and subsequently updated in the InO-filter and SU if not found.

A. Hash Functions

The space efficiency of Bloom filters relies on the universality of the hash functions, i.e, their equal probability of hashing into the entire filter independent of data. We use a class of universal hash functions, H_3 as described in [11], that have been found suitable for hardware designs. Specifically if d_i represents the ith data bit $\forall i \in [1 \ldots n]$ and q_j^l represents the jth bit of a random number in the range $[1 \ldots m]$ then the hash function i can be given as

$$H_i =< h_1^i, h_2^i, h_3^i, \ldots, h_n^i >$$

where

$$h_i^l = q_1^l.d_1 \oplus q_2^l.d_2 \oplus \ldots \oplus q_n^l.d_n$$

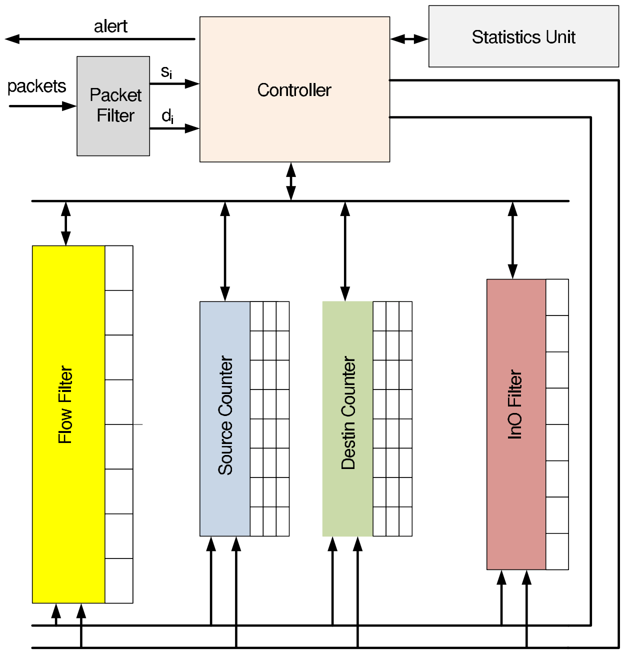

Fig. 4. System Architecture

B. Multi-port Embedded Memories

As discussed earlier, the Bloom filters work by using k hash functions. These hashes can be done serially on a single big chunk of memory or performed in parallel on multiple smaller memories. The effect of latter is a negligible change in the accuracies as discussed in [12]. We therefore employ multiple-port embedded memories that are partitioned into smaller chunks, with each having the flexibility to perform an independent read/write operation.

V. Analytical Evaluation Framework

The design presented in the last section consists of a number of Bloom filters and CBFs that rely on one another for the various statistics being reported. The accuracy of the individual statistic is thus tied with the accuracies of various filters/CBFs that lie in the path for the statistic being recorded. For instance, to identify correctly a new InO node, the incoming flow must correctly pass through the conversation, degree and InO filters. A larger filter size at the InO filter may thus not yield as much improvement in InO accuracy as if the same resources were invested in degree CBFs. Given a limited resource budget, it is therefore logical to ask where the design needs to invest more resources to increase the overall system accuracy. This section presents an analytical framework that can help in answering these questions.

We begin by noting that the conversation filter works in isolation. Thus, its accuracy is entirely dependent on its false positive rate. Let m_i, n_i and k_i represent the filter size, number of discrete input keys and hash functions for the conversation filter ($i = 1$), degree CBFs ($i = 2$), and InO filters ($i = 3$) respectively. Then the accuracy of correctly identifying a new flow is given by

$$A_c = 1 - f_{p1} = 1 - (1 - e^{-n_i k_1 / m_1})^{k_1}$$

The degree count accuracy (A_d) is however tied with the accuracy of conversation filter along with accuracies of the CBFs. An address may result in a false degree increment whenever the conversation filter gives a false edge or even with a correct edge but there are inaccuracies in the corresponding counter. Therefore,

$$A_d \quad = \quad 1 - \{f_{p1} + t_{p1} f_{p2}\}$$

Similarly, a new node fails to get counted if its flow gets matched with an already logged flow in the conversation filter or if any of the two CBFs falsely match it with another other node within them. Therefore the node count accuracy can be given as

$$A_n \quad = \quad 1 - \{f_{p1} + 2t_{p1}.f_{p2}\}$$

Finally, the InO count is falsely incremented whenever an incoming node matches the presence test in both the CBFs, with at least one of them being false. The incoming node has to pass through the conversation filter and also to be declared not present by the InO filter. Assuming independence, this can be given as

$$A_{ino} \quad = \quad 1 - \{|(t_{p1} - fp1)(2f_{p2}t_{p2} + f_{p2}f_{p2})(t_{n3})|\}$$

The question of where to allocate more resources can now be answered. We first need to establish sufficiently low false rates in all the filters/CBFs. However, if one has to decide where to put extra resources, the focus of accuracy may need to be taken into account. If more attention is to be given to a high node/degree count accuracy, a low false positive rate of the conversation filter has to be ensured as a high false positive at the filter may prevent new nodes from reaching the CBFs. However, if the focus is InO accuracy, significant resources also have to be allocated to the CBFs as they are involved in both false and positive evaluations of the conversation filter.

VI. Empirical Evaluation

A. System Throughput

We first explore the throughput of our design. The design's throughput is data-dependent because of data-dependencies in edge and conversation filtering. For instance, a very restrictive edge-filter may only let a few packets to be processed yielding a very high system throughput. Similarly, as the flow-filter gets populated, more and more incoming flows will match in the filter and therefore not forwarded to later stages, resulting in an increase in the system throughput.

We test the system throughput using a basic single packet HTTP filter that ensures maximum number of packets passing through the flow-filtering stage. Further, we only send HTTP packets in the trace, thus effectively bypassing positive effects on throughput by the initial edge-filtering. The results so obtained are plotted in Figure-5. As expected, the throughput is steadily increasing with an average throughput of 7.4 million packets/second. We stress that this is a conservative figure and in practice, the throughput from our system should be much better in a mixture of different types of packets.

B. Accuracy Tuning

We now use the observations from the previous section in tuning the sizes of different filters in the design. We begin by noting that the flow-filter acts independently and therefore its false positive rate is entirely dependent on its own parameters (m_1 and k_1) and number of input flows (n_1). We make use of equation-3 to compute number of hash functions that guarantee a sufficiently low false positive rate. We chose the value of $k_1 = 8$ (yielding $f_{p1} = 0.0039$). The number of hash-functions also represent the number of partitions of the filter as discussed in section-IV-B. We next turn our attention to evaluating the size of individual partitions. This is done by using equation-2 that gives us a value of $m \approx 14K$ for $n_1 = 10k$ flows. Since the partitions on the Xilinx FPGA only come in regular sizes of power of two, we fix m being $16k$. For simplicity, we stick with 8 hash functions (and thus partitions) for the remaining filters in the design.

Table-III represents a set of system configurations for exploring the system performance using different individual filter sizes along with the resulting total system RAM requirements. For accuracy measurements, we disable the initial packet filter since the number of filtered packets were too few to have a stressful system accuracy evaluation. Like throughput, the accuracy of the filters is also highly dependent on number of packets that are already logged in the filters. Figure-6 and Figure-7 present the accuracy plots with number of packets in the system. The final accuracy values are also tabulated in Table-III.

The plots demonstrate the dependency of accuracy of our design with number of packets seen so far. The first configuration of Table-III has minimal memory requirements but fares poorly in accuracies, in particular the InO accuracy. We therefore use our insights from previous section and double the size of CBFs in the second configuration. The change results in more than $3x$ improvement in InO accuracy, despite reduction in the sizes of other filters. We next focus on increasing node count accuracy. The third configuration achieves this by doubling the flow filter size. To see the effects on node count accuracy due solely to the size of the CBFs, we go back to the second configuration and increase the counter size in the fourth configuration. We note that the change has less effects on node count accuracy than to the InO accuracy, thereby confirming to our earlier observations from previous section. The other configurations further explore the design space. We select configuration $C - 8$ as our final system configuration, offering a high amount of accuracy with a decent area-budget.

No.	Conv Filter x8 (Kb)	InO Filter x8 (Kb)	Degree Ctrs x2x8x8 (Kb)	Total (Kb)	Node Count Accuracy	InO Count Accuracy
C-1	16	16	2	512	71.78	27.35
C-2	8	8	4	640	88.31	85.75
C-3	16	8	4	704	92.14	81.07
C-4	8	8	8	1152	90.13	99.18
C-5	16	8	8	1216	94.46	100
C-6	32	8	8	1344	96.95	99.34
C-7	32	4	8	1344	99.95	99.34
C-8	32	1	8	1312	96.95	99.34
C-9	32	16	16	2432	99.98	99.67

TABLE III
SYSTEM EVALUATION

Metric	Value	Device Utilization
IO Pins	69	14.4%
Number of occupied Slices	754	10%
Number of Block-RAMs	44	73%
Clock (MHz)	107	-

TABLE IV
IMPLEMENTATION RESULTS

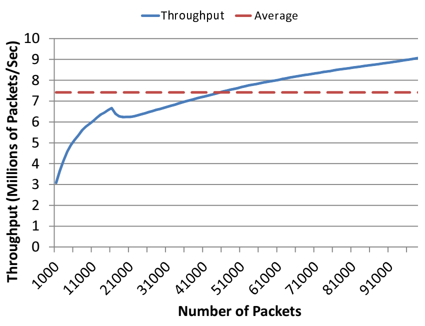

Fig. 5. System Throughput

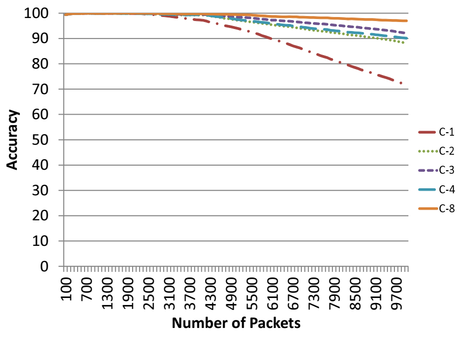

Fig. 6. Node Count Accuracy

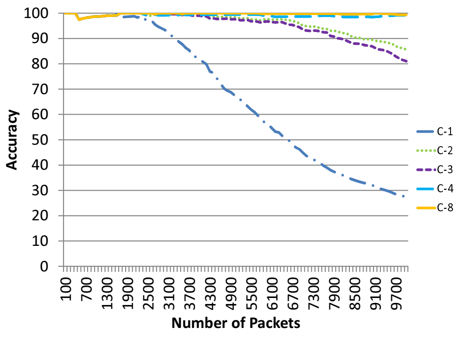

Fig. 7. InO Count Accuracy

C. Prototype Implementation

We prototyped our design on Virtex-5, $XC5VLX50t$ device using Xilinx ISE 11.1. Table-IV shows the prototype results. The device incorporates 60, $32k$ Block-RAMs that were used to map the Bloom filters/CBFs in the design. It can be noticed that bulk of the device consumption is due to the filters whereas the associated logic itself takes quite minimal die-resources. This shows that the device can be further utilized in implementing more richer set of edge-filters and alarm rules for various other kinds of applications using the same TDMs.

VII. CONCLUSION

The paper presented a novel FPGA based solution for real-time collection of metrics involving Traffic Dispersion Graphs (TDGs). The TDGs have been shown to offer insights for network traffic analysis but there has not been a concentrated effort in their real-time online evaluation. We analyzed the metrics presented in the literature with the focus on their online evaluation. We used our analysis in a Bloom filter based solution and evaluate its efficiency using analytical and empirical means. The presented solution enabled real-time online evaluation of TDGs, processing 7.4 million packets/second for a TDG comprising of $10k$ flows at very high accuracies of over 96%.

REFERENCES

[1] J. Erman, A. Mahanti, and M. Arlitt, "Byte me: a case for byte accuracy in traffic classification," in *MineNet '07: Proceedings of the 3rd annual ACM workshop on Mining network data*, 2007, pp. 35–38.
[2] T. Karagiannis, A. Broido, M. Faloutsos, and K. claffy, "Transport layer identification of P2P traffic," in *IMC '04: Proceedings of the 4th ACM SIGCOMM conference on Internet measurement*, 2004, pp. 121–134.
[3] M. Crotti, M. Dusi, F. Gringoli, and L. Salgarelli, "Traffic classification through simple statistical fingerprinting," *SIGCOMM Comput. Commun. Rev.*, vol. 37, no. 1, pp. 5–16, 2007.
[4] A. Lakhina, M. Crovella, and C. Diot, "Mining anomalies using traffic feature distributions," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, 2005, pp. 217–228.
[5] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "BLINC: Multilevel traffic classification in the dark," in *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, 2005, pp. 229–240.
[6] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese, "Network monitoring using traffic dispersion graphs (TDGS)," in *Proceedings of the 7th ACM SIGCOMM Internet Measurement Conference*, 2007, pp. 315–320.
[7] Y. Jin, E. Sharafuddin, and Z.-L. Zhang, "Unveiling core network-wide communication patterns through application traffic activity graph decomposition," in *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, 2009, pp. 49–60.
[8] M. Iliofotou, H.-c. Kim, M. Faloutsos, M. Mitzenmacher, P. Pappu, and G. Varghese, "Graph-based P2P traffic classification at the internet backbone," in *INFOCOM'09*, 2009, pp. 37–42.
[9] M. Iliofotou, M. Faloutsos, and M. Mitzenmacher, "Exploiting dynamicity in graph-based traffic analysis: Techniques and applications," in *ACM CoNEXT*. New York, NY, USA: ACM, December 2009.
[10] F. Khan, L. Yuan, C.-N. Chuah, and S. Ghiasi, "A programmable architecture for scalable and real-time network traffic measurements," in *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008, pp. 109–118.
[11] M. Ramakrishna, E. Fu, and E. Bahcekapili, "A performance study of hashing functions for hardware applications," in *In Proc. of Int. Conf. on Computing and Information*, 1994, pp. 1621–1636.
[12] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood, "Deep packet inspection using Parallel Bloom Filters," *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004.