A Single-FPGA Architecture for Detecting Heavy Hitters in 100 Gbit/s Ethernet links

Jose Fernando Zazo*, Sergio Lopez-Buedo*†, Mario Ruiz†, Gustavo Sutter†
*NAUDIT HPCN
Calle Faraday 7, 28049 Madrid, Spain
†High-Performance Computing and Networking Research Group, Universidad Autonoma de Madrid
Ciudad Universitaria de Cantoblanco, 28049 Madrid, Spain

Abstract—**In network traffic monitoring, a very important analysis is to find heavy hitters. That is, finding those flows that use most resources in a given network link. This information can be very useful for security or traffic management purposes. Though this analysis might seem easy to implement, since it is essentially based on counting, the fact is that doing it at 100 Gbit/s rates is far from trivial. In 100 Gbit/s Ethernet (100 GbE), up to 148 million packets per second can be received, thus making it very difficult to parse packets and maintain counters at such rate. In this paper, we leverage the integrated 100G Ethernet Subsystem available in Xilinx UltraScale devices to implement a heavy hitter detector for 100 GbE in a VCU108 evaluation kit. Thanks to the integration of the Count Sketch algorithm with a priority list and a network packet parser, the proposed architecture is able to work at line rate for average packet sizes bigger than 215 bytes. The work presents a theoretical analysis of the error, as well as the technical details of the proposed solution. The implementation has been validated using real-world traces, obtaining an average error of 1.29%.**

I. Introduction

Counting elements is an everyday activity. It is the basic brick for generating statistics, monitoring physical phenomena or studying problems in a diverse range of fields. Computer networks are no exception. Commercial switches and routers use counters to provide aggregated information about the number of packets, the number of connected users or the bandwidth usage. Such simple metrics are however very important to understand the networking infrastructure and ensure its proper functioning.

Solutions for counting that involve several heuristics are not unusual. Sampling and hashing are two well-known candidates in a race for achieving a trade-off between accuracy and resource consumption. Static structures, in terms of memory usage, are always beneficial for hardware developments. Sketches feature such static structures and belong to the the group of the hashing techniques. Initially proposed in 1995 [1], they are an elegant alternative composed by a bidimensional array of d rows and w columns, which are defined during the design of the structure.

Drawbacks arise when our concerns become somewhat more exigent, when we move from a mere counter to item classification. "Which ones are the most influential?" or, "what are the current trends?" are the kind of questions that this paper aims to solve. The goal, from a network analytics perspective, is to determine users that generate the most of the traffic or finding out which servers are more heavily loaded. However, determining the most frequent elements in a dataset is not a straightforward task and it is even a more difficult when working at line rate in 100 GbE links. At such speed, items cannot be inspected more than once, and the packet rate of a fully loaded link can reach 148 million packets per second.

Network analytics is not the only vibrant topic where determining the number of occurrences of an item is a key aspect. Natural language processing or webpage indexing by a search engine are other instances where the huge volume of information cannot be counted at runtime with a limited memory consumption, unless a fair equilibrium between accuracy and resource utilization is established.

M. Charikar [2] suggested a software approach when counting the most frequent elements: integrate a sketch structure in combination with a fixed-size heap. For the purpose of this work, tree-based structures, though they are optimal for software implementations, may not be so alluring for a hardware design. Indeed, implementing a priority queue (PQ) is a demanding task itself [3]. Fortunately, the size of the ranking list is generally in the order of tens of elements, so certain simplifications to the design fit in the scope of this application.

The reduction in memory requirements achieved by limiting the size of the ranking list makes the design ideal for FPGA implementation. The Count Sketch (CS) algorithm as well as a PQ can fit on BRAM memories, thus obtaining a high speed up because of the tangibly diminution of the latency when compared with conventional memories. Additionally, dedicated hardware can fully exploit the parallelism of the algorithms (multiple computations of hash functions or simultaneous comparisons to look for the greatest values). These two facts allowed us to scale up the framework to the demanding scenario of 100 GbE network analytics.

II. Related Work

The action of finding the top-n elements is solvable by counter-based techniques. Using a particular counter for every possible candidate in the data stream, and sorting the different counters by value, is an efficient solution when the requirements of the problem are well defined. One of the main drawbacks of this approximation is that the total number of counters increases linearly with the number of monitored

elements. Another drawback is that if the number of counters is statically defined at design time, and an unexpected element is observed, the design will discard its associated information, even if it is one of the top-n. In [4], D. Tong et al. propose an FPGA-based application that exploits the counter-based technique. Up to 128000 concurrent flows can be concurrently managed, at a maximum throughput of 84 Gbps.

Refinements to the methodology are possible, this is the case of the Lossy Counting algorithm [5], which is a popular counter-based routine. In this algorithm the elements with a low count are periodically removed from the table of results. The idea is natural and intuitive: the input data stream is divided in windows and, after each window, all counters are decremented by a unit. If the associated frequency count of an element (originally initialized to a conservative value) has reached the value of 0, such element will no longer be taken into account, as it is not considered to be part of the top-n list. The process continues considering exclusively the smaller subset for the next epochs.

The main critic associated to counter-based approaches is the intensive use of memory. This drawback makes it very difficult to apply these techniques in a real-world environment where the elements of the input data stream may belong to a very diverse alphabet. Alternatively, sampling of the incoming elements is a solution exploited by commercial network devices. The Sticky Sampling algorithm [6] is an example of this technique. A set counters (associated to the elements to be inspected) are created by sampling with a certain probability r. Once that an element has been marked as monitored, the system keeps its exact count. Every time that the sampling rate, r, is recomputed, the frequency of the item is readjusted. If it becomes 0 in a readjustment (in the same manner that the Lossy Counting algorithm), the element stops being considered as relevant. The final result will be statistically bounded. However, because not all the elements of the input data stream are considered, it may lead to a situation where, with a non-null probability, an element belonging to the top-n is not monitored.

Sketch-based techniques, on the other hand, do not limit its estimation to a subset of elements. D. Tong et al [7] propose a FPGA-based implementation of the Count-Min Sketch [1] for the estimation of heavy hitters. The Count-Min Sketch relies on the update of d counters every time that an element is observed. The counters are chosen by d pairwise independent hash functions and the operation is as simple as incrementing the previous value with the new observation. To alleviate the caused effects by possible collisions, the estimation for the query point of an item will be the minimum value of the d counters that are referenced by the hash functions.

A problem of the Count-Min Sketch is that the estimator is biased. In other words, it is expected that the algorithm, without any kind of optimization, will asymptotically return an overestimation. Some improvements to the original algorithm were proposed at [8] with the introduction of conservative updates. The underlying idea is to update an entry only if the current value of the counter is not greater than the estimation.

Other kind of sketches, which are not based on positive increments, are also available. This is the case of the CS structure [9]. A sign will be associated with an update, so the increment could either be positive or negative. In this way, when a collision occurs, it is expected that half of the times it will increment the counter, while the other half it will decrement the counter, thus canceling the effect of the collision. That is, the CS is an unbiased estimator, so the difference between the estimation and the real value will tend to zero. For further details and discussions about sketches, in [10] a whole comparison between 10 different sketches is introduced in the context of natural language processing.

III. Methodology

A. Intuition

The action of finding heavy hitters in a network is a challenge by itself. In this paper we classify as heavy hitters those network flows that have transmitted the biggest quantity of packets. However, the proposed method can be easily extended to considering bytes per flow instead of number of packets. The main problem is the dimensionality of the origin space: There are 2^{32} different elements if the IPv4 source address is used as the identifier of a network flow. Given the impossibility of coping with the original space, a probabilistic data structure is applied for its projection. Such projection will insert an error that can be bounded. It means that with probability $1 - \delta$ we will be sure that the error is less than a certain value, which depends on the parameter ε. Terms δ and ε are detailed in Section III-D.

Once that the original space has been reduced, counting the total number of occurrences and choosing the highest values are the next steps to perform. The action of counting the elements in the stream of packets is achieved by the application of the CS structure whilst the implementation of a priority queue will provide the ranking list of the most significant network flows. Both concepts are detailed below.

B. CS structure: update process

The CS structure is able to provide an estimator of the total number of occurrences of an element in a data stream. More generally, the utilization of the sketch is divided into 3 differentiated steps: projection of an element, update of counters and a point query. Let denote by Ω the different possible values for an element in the data stream. Let denote by T a table of $d \times w$ counters that have been initially set to 0 and let be $IP \in \Omega$ (the source IP address of an incoming packet).

For such element IP, the workflow consists on:

- Computing $2d$-wise independent hash functions:

  $$F := \{f_i : \Omega \to [1, w]\}_{i=1}^{d}, G := \{g_i : \Omega \to \{-1, +1\}\}_{i=1}^{d}$$

  The family F maps the original space into a reduced space of cardinality w. Meanwhile, the set of functions G maps the original space into the space $\{-1, +1\}$.

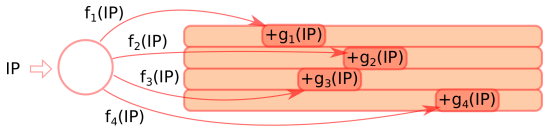

Fig. 1: Increment in the Count Sketch Structure

- Updating d counters in the sketch:

$$T_{i,f_i(IP)} = T_{i,f_i(IP)} + g_i(IP), i = 1, ..., d$$

The CS structure is ideal for hardware implementations, not only because its memory requirements are static, but also due to its parallelization capabilities whilst computing the hash functions and updating the counters. It uses a memory of size $d \times w \times l$, where d and w are the depth and width of the counting table, and l the size of the counters. The error that is made with the CS depends on the selected values for d and w (assuming a value of l that does not overflow), and it is evaluated in Section III-D.

Fig. 1 illustrates the procedure of updating an entry for a new arriving frame. Once that the source IP address is known, d counters are incremented or decremented by one (according to the family of functions g_i) at positions referenced by functions f_i.

C. CS structure: point query

Given an element $IP \in \Omega$ and the previously defined sketch structure, the estimation is obtained as: $median\left(\{g_i(IP)T_{i,f_i(IP)}\}_{i=1}^{d}\right)$. So, the approximated value of transmitted packets by a source IP address lies in the computation of the median of d values. This operation depends on sorting the whole set of elements and selecting the value that divides the sorted list into two halves (if d is odd) or that leaves half of the elements at the left side (if d is even).

In order to tackle this problem in hardware, bitonic sort is a parallel sorting network that will help in this process. With $O(n \log^2 n)$ comparators, this algorithm is not optimal in software (for instance, mergesort just needs $O(n \log n)$). But the main advantage of bitonic sort is that comparisons are not data dependent, and therefore, the algorithm is suited for a parallel hardware implementation.

D. Selection of the optimal values

The goal of this point is to quantify the compromise between resource consumption and acceptable errors in a real-world deployment. The parameters to determine are d and w (depth and width of the Sketch), given a maximum acceptable error and the probability of this error happening. Setting $d = \log \frac{4}{\delta}$ and $w = O\left(\frac{1}{\varepsilon^2}\right)$, we can be sure (according to [2]) that the error made is bounded by εn (where n is the total number of observations) with probability at least $1 - \delta$. This approximation for the error can be indeed refined in terms of the second moment of frequency, $F_2 = \sum_{i \in \Omega} o(i)^2$, where $o(i)$ returns the total number of occurrences of the IP address i in the dataset. The CS structure assures that the error is less or equal than $\varepsilon\sqrt{F_2}$, whose estimation could be deduced at runtime [11] in order to have a finer control of the error made.

δ	ε	$d = \log \frac{4}{\delta}$	$w = O\left(\frac{1}{\varepsilon^2}\right)$	Error [Packets]
0.001	10^{-3}	8.29	10^6	≤ 148.8095
0.001	10^{-6}	8.29	10^{12}	≤ 0.1488
0.01	10^{-3}	5.99	10^6	≤ 148.8095
0.01	10^{-6}	5.99	10^{12}	≤ 0.1488
0.05	10^{-3}	4.38	10^6	≤ 148.8095
0.05	10^{-6}	4.38	10^{12}	≤ 0.1488

TABLE I: Values of d and w for different combinations of ε and δ, and maximum error in a 100 GbE link during 1 ms

In TABLE I, some combinations of δ and ε are presented, together with the corresponding values of d and w and the errors made during 1 ms of monitoring. A bigger probability (smaller δ) requires more rows in the table (bigger d) and therefore, more memory. Obviously, fractional values for d only make sense during the calculation of δ and ϵ; integer values of d will be used for dimensioning the counter table. Regarding the error made, the formula is εn where in this case n is the maximum number of packets that can be received during 1 ms, considering the worst case of a link fully loaded with minimum-size packets (148 MPPS).

E. Priority queue (PQ)

Once that the estimation of the number of transferred packets by a source IP address is known, the next step is to check if such number belongs to the largest observed values. The largest values will be stacked at the first positions of the list whilst the smaller ones will disappear as the list gets fully populated. The algorithm used by the PQ is:

Algorithm 1 Tracking the most significant IPs

Input: $(IP_a, p_a), ranking$
insertion=length(ranking)
for $i = length(ranking)$ to 1 **do**
  if $((IP_a, p_a) > ranking[i])$ **then**
    $insertion = i; tmp = ranking[i]$
  end if
end for
for $i = insertion + 1$ to $length(ranking)$ **do**
  $tmp = ranking[i+1]; ranking[i] = tmp$
end for
$ranking[insertion] = (IP_a, p_a)$

The previous algorithm can be parallelized at two different points: during the search of the associated position and while shifting the elements.

IV. Hardware Design Overview

Workflow starts with the parsing of network packets. The design inspects the EtherType field of the packet and verifies that it is indeed an IPv4 packet. If a match occurs, the source IP address is propagated to the next component, the CS, where estimations are computed. Following, the estimations from the CS are used for populating the PQ, which only keeps the

DRAFT
October 31, 2017, 15:15

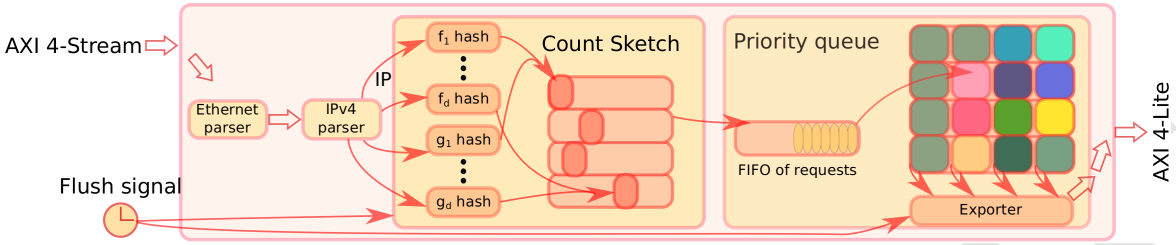

Fig. 2: Block design of the project

most active elements. Independently of the incoming traffic, a periodic flush signal is asserted. This signal forces the reset of the CS and PQ tables. However, before completely removing the top IP addresses, the current ranking is exported using an AXI4-Lite interface. Here, the key idea is that the aggregated information is easily processable by software (or a third-party element), and periodically resetting the counters will reduce the probability of overestimating the number packets per source IP address. Note that the specified boundary at Section III-D was dependent on the total number of received packets, and up to 148 million packets can be received per second in a fully loaded 100 GbE link. The full picture is illustrated in Fig. 2.

A. Network packet parser

In [12], an automated compiler for the generation of network filters is detailed. This work uses a packet parser generated with that compiler to verify if the Ethernet frame encapsulates an IPv4 packet, and if so, to extract the IP source address. In this way, the HDL code for the packet parser is automatically generated, thus reducing significantly the development time. An additional benefit is that the parser could easily be extended in order to support MPLS or Ethernet VLANs.

B. CS structure: table of counters

The size chosen for the table is $d = 8, w = 2^{20}$. Considering these values for d and w, error is less than 0.1% with a probability higher than 99,86%. The table needs 8 rows of 2^{20} entries, which is small enough to fit in 256 BRAMs, therefore avoiding the latency penalties associated with bigger memories.

C. CS structure: selection of hash functions

The family of hash functions that has been chosen for the experiment is H_3 [13]. Given a source IP address k, for $m_i \in \{0,1\}^{32\times 20}, i \in 1,...,d$, the family hash functions $F = \{f_i\}_{i=0}^{d}$ is defined as:

$$f_i(k) = k_1 \wedge m^i_{(1,\cdot)} \oplus ... \oplus k_{32} \wedge m^i_{(32,\cdot)}$$

Where $\wedge$ and $\oplus$ respectively represent the logical AND and XOR operators. The family of functions G is analogously defined, with the only difference that random matrices are selected from the space $\{0,1\}^{32}$:

$$g_i(k) = k_1 \wedge m^i_1 \oplus ... \oplus k_{32} \wedge m^i_{32}$$

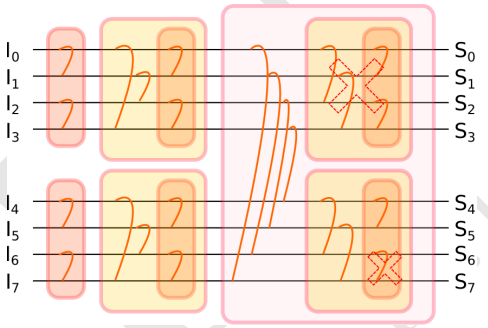

Fig. 3: Bitonic implementation

In this paper, matrices have not been verified for the pairwise independence, this is an issue that will be tackled in future developments.

D. CS structure: computation of the median

Given a network identifier and the family of hash functions F, a total of d counters are referenced. Estimation is computed as the median of the d counters; Fig. 3 represents the bitonic network that has been used to obtain the median. At the left side of the network the unsorted sequence of numbers acts as the input. The output is the same sequence which has been permuted according to the values of the elements. S_0 will be the minimum value while S_7 is the maximum one. The wires that appear as interconnected should be interpreted as a possible swap of elements. Every time that two black wires are interconnected, the logic will check if the element at a higher position is greater than the other one. If such is the case, the values of both wires are swapped.

Because just the signal S_4 is strictly necessary for computing the median, some comparators in the last level of the hierarchy of the design can be removed (red cross).

E. CS structure: interconnection of the stages

The architecture of the CS is fully pipelined and the associated latency is:

- One clock cycle for the computation of the family F and G of hash functions. All functions are computed in parallel.
- Four cycles for updating the table of the sketch:
  1) Read the previously stored value from the BRAM.
  2) Computation of the median.

PREPRINT Friday 3rd November, 2017, 15

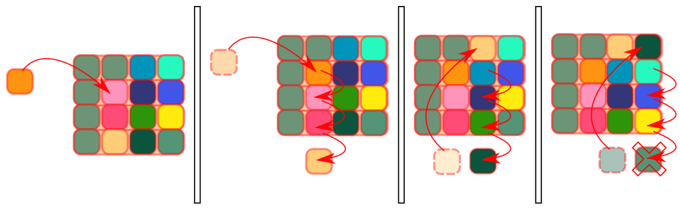

Fig. 4: Priority queue implementation

3) Computation of the next value.
4) Write back the updated information to the BRAM, and generation of an update request to the PQ.

Additionally, if two consecutive requests write to the same addresses, there occurs a data hazard in the pipeline that causes the loss of the first request. In order to avoid such undesirable behavior, a data forwarding path to step number 3) have been implemented from step 4).

F. Priority queue

The PQ contains the tag of the CS elements (32 bits corresponding to the source IPv4 address) and the estimation of the number of packets (18 bits). Additionally, an extra bit is necessary to indicate if a position is valid in the queue or, on the other hand, such position is free. At the very first, the PQ will set the valid bit to 0 in each position. Once that this reset has been done, the design will wait for next update of the sketch. The estimation of the number of packets and the associated IP source address are the inputs to the PQ. The PQ table is updated every time that a new pair arrives, so that elements in the table are always ordered by value.

The PQ table has been designed as a parallel memory where multiple cells can be accessed concurrently. In our reference design, the dimensions of the PQ table matrix Q are 4×4. Elements of Q are sorted, so given an index $(i,j) \in \mathbb{N}^2$, $Q_{(i,\cdot)} > Q_{(i',\cdot)} \forall i' > i$ and $Q_{(i,j)} > Q_{(i,j')} \forall j' > j$. That is, the largest element will be always at the position $Q_{0,0}$.

Fig. 4 shows the insertion of an element in the PQ table. Here the table is completely full and, at the same time that the key is looked up, a candidate position for its insertion is determined. Once that the new position for the element is known, all the elements in the same column are shifted to the next row. The element at the bottom row will be inserted as the first element of the next column. This process is next repeated for every column. It is important to note that shifts can also go the left, because an update may also decrement the observed value of an element (due to errors associated to the sketch structure), and therefore its current position will be deflated.

Regarding the physical implementation, every row of the PQ table corresponds to a different BRAM. Columns correspond to different addresses of the row BRAM. Typically, the size of the ranking list will be relatively small, but this implementation will anyway allow to scale up the design (with the penalty of increasing the latency while looking for an element or while carrying out the shifts). Querying and insertion time for any particular element is in the order of $O(n)$, where n is the total number of elements to be monitored in the ranking list. In our case of the top list has 15 elements, we instantiate 4 parallel BRAMs and the worst latency while querying for a particular element is 4 clock cycles, while the worst latency for insertion is 8 clock cycles. On top of the previous latencies, 4 additional clock cycles are always added, associated to the computation of the next addresses to be written once that the element has been located inside the table.

Resources	CS	PQ	Total
LUT	728	603	1431 (0.12%)
LUTRAM	17	0	17 (0.01%)
FF	1482	419	2159 (0.09%)
BRAM	256	4	261 (12.08%)

TABLE II: Categorized FPGA occupation.

V. Results

The block diagram of the whole design, which integrates the frame parser, the CS structure and the PQ, is depicted in Fig. 2). The input to the design is an AXI4-Stream interface that comes from the 100 GbE MAC/PCS core. On the output side, the ranking list of the top-15 source IP addresses is made available via a slave AXI4-Lite. The design has been successfully implemented in a Xilinx Virtex Ultrascale VCU108 reference board, and it is able to process more than 26855731 packets per second from the 100 GbE network interfaces available in the VCU108 board.

The previous value for the number of packets processed per second corresponds to an average packet size greater than 450 bytes. According to the traces provided by CAIDA [14], this limitation should not be a major drawback for a backbone link. Actually, statistics for CAIDA traces from 2008 to 2016 show that the average packet size is in all cases bigger than 700 bytes. However, alternative configurations for the PQ would allow increasing the number of packets processed per second, since the main cause of the packet rate limitation is the latency of the PQ operations. For example, increasing the PQ memory from 4 to 15 BRAMs will allow to improve parallelism and thus reducing the average packet size limit to 215 bytes.

An empirical evaluation of the proposed CS/PQ algorithms have been carried out on 13 different network traces. The first 12 were provided by CAIDA and correspond to the backbone link Equinix-Chicago. Each trace was capture during one hour of a different month during of 2009. The last trace corresponds to a web server, and was obtained from the datacenter of a big Spanish company. For this evaluation, the analysis of the traces using tshark has been considered as the ground truth. In all traces but one the ranking list obtained by the estimator matches that obtained with tshark. For the only case where there was a mismatch, what it happened is that the 16th and 15th elements in the ranking list were swapped.

Fig. 5 shows the errors made in the estimation of the number of packets per flow. 4 out of the 12 CAIDA traces are represented, as well as the datacenter trace. For analysis, traces were split in chunks of 1,000, 10,000, 100,000, 200,000

DRAFT

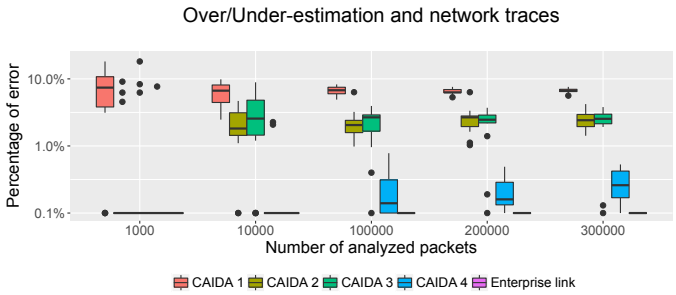

Fig. 5: Empirical results of the algorithm for different network traces

and 300,000 packets. Errors in CAIDA traces are relatively big because the number of packets per flow is small. For example, the biggest error detected (18.18%) corresponds to CAIDA 1 trace split in chunks of 1000 packets. In that case the actual number of packets for the 13th top flows is 11, but the estimation is 13 due to 2 collisions. For the datacenter traces, where the number of packets per flow is much bigger, errors are 2 orders of magnitude smaller. As a summary, average error was 1.29%, the median 1.23%, the 95 percentile 1.99% and the 99 percentile 2.47%.

It can also be noticed in the figure that the variance of the error diminishes as the number of packets per analyzed chunk increases. The reason for this behavior is that when the number of analyzed packets is big, the number of collisions will also be big but the errors caused by these collisions will tend to be negligible. The last affirmation is motivated because of the insignificance of the errors in comparison with the contribution of heavy hitters. Finally, the reason for the outliers in the figure lies in the fact that matrices for the F, G family of hash functions have not been checked for pairwise independence.

VI. Conclusions

This paper presents a novel architecture based on FPGAs for ranking the elements in a data stream. As opposed to other solutions, the estimator used in this work is not biased, it does not require multiple inspections of the incoming data, and it does not use sampling. The use case that have been selected is ranking the source IP addresses in order to find heavy hitters in a 100 GbE network link, that is, finding those IP addresses that generate more packets. However, the proposed approach is totally adaptable to other metrics (most accessed workstations, most used transport protocol ports, etc.) and to other fields (such as the detection of the most repeated words in a text document). An empirical validation of the method, using CAIDA backbone traces as well as a datacenter trace, shows that the average error made is 1.29 %

The design has been successfully implemented on a Xilinx UltraScale device, using a VCU108 evaluation kit. It meets the stringent timing constraints required by the 100G Ethernet Subsystem of UltraScale devices (322.265625 MHz clock), and it is able to work at line rate provided that the average packet size is bigger than 215 bytes. Future work will include a detailed study of the pairwise independence of hash functions in order to improve the quality of estimations, and a reduction in the latency of operations in order to increase the maximum number of packets that can be processed per second.

Acknowledgment

This work was partially supported by the Spanish Ministry of Economy and Competitiveness and the European Regional Development Fund under the project TRÁFICA (MINECO/FEDER TEC2015-69417-C2-1-R), and by the European Commission, under the dReDBox project (grant agreement No. 687632) and METRO-HAUL project (grant agreement No. 761727), both of the H2020 programme.

References

[1] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *Journal of Algorithms*, vol. 55, pp. 58–75, Apr. 2005.

[2] G. Cormode and M. Hadjieleftheriou, "Methods for Finding Frequent Items in Data Streams," *The VLDB Journal*, vol. 19, pp. 3–20, Feb. 2010.

[3] R. Hui, "Priority queue architecture for supporting per flow queuing and multiple ports," Aug. 2004. U.S. Classification 370/412, 370/395.4; International Classification H04L12/56; Cooperative Classification H04L47/2416, H04L47/50; European Classification H04L12/56K, H04L47/24B.

[4] D. Tong and V. Prasanna, "Online heavy hitter detector on FPGA," in *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Dec. 2013.

[5] G. S. Manku and R. Motwani, "Approximate Frequency Counts over Data Streams," in *Proceedings of the 28th International Conference on Very Large Data Bases*, VLDB '02, (Hong Kong, China), pp. 346–357, VLDB Endowment, 2002.

[6] G. S. Manku, S. Rajagopalan, and B. G. Lindsay, "Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, (New York, NY, USA), pp. 251–262, ACM, 1999.

[7] D. Tong and V. Prasanna, "High Throughput Hierarchical Heavy Hitter Detection in Data Streams," in *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pp. 224–233, Dec. 2015.

[8] A. Goyal and H. Daum, III, "Approximate Scalable Bounded Space Sketch for Large Data NLP," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, EMNLP '11, (Stroudsburg, PA, USA), pp. 250–261, Association for Computational Linguistics, 2011.

[9] M. Charikar, K. Chen, and M. Farach-Colton, "Finding frequent items in data streams," *Theoretical Computer Science*, vol. 312, pp. 3–15, Jan. 2004.

[10] A. Goyal, H. Daum, III, and G. Cormode, "Sketch Algorithms for Estimating Point Queries in NLP," in *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*, EMNLP-CoNLL '12, (Stroudsburg, PA, USA), pp. 1093–1103, Association for Computational Linguistics, 2012.

[11] N. Alon, Y. Matias, and M. Szegedy, "The Space Complexity of Approximating the Frequency Moments," in *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, (New York, NY, USA), pp. 20–29, ACM, 1996.

[12] J. F. Zazo, S. Lopez-Buedo, G. Sutter, and J. Aracil, "Automated synthesis of FPGA-based packet filters for 100 Gbps network monitoring applications," in *2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pp. 1–6, Nov. 2016.

[13] M. V. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Transactions on Computers*, vol. 46, pp. 1378–1381, Dec. 1997.

[14] "The CAIDA UCSD Statistical information for the CAIDA Anonymized Internet Traces."

PREPRINT
Preliminary Version