Hardware Modules for Packet Interarrival Time Monitoring for Software Defined Measurements

Racyus Pacífico[1], Pablo Goulart[2], Alex B. Vieira[3], Marcos A. M. Vieira[2], José Augusto M. Nacif[1]
[1]Science and Technology Institute, Universidade Federal de Viçosa, Campus UFV-Florestal
[2]Computer Science Department, Universidade Federal de Minas Gerais
[3]Computer Science Department, Universidade Federal de Juiz de Fora
Email: {racyus.pacifico, jnacif}@ufv.br, alex.borges@ufjf.edu.br, mmvieira@dcc.ufmg.br

***Abstract*—Measurement and tracking have crucial roles in Software-Defined Networks (SDNs). Unfortunately, most of procedures and techniques to perform measurements and monitoring tasks are implemented in software at network end-hosts. Despite the large use, a software based approach generates imprecision, high costs, and makes monitoring more difficult. In this paper, we extend OpenFlow switch to implement a measurement architecture for SDN networks. Our system performs measurements in a simple and scalable way without depending on end-hosts. It allows monitoring the performance at the granularity of flows. Moreover, our system also enables software-defined measurements since the controller can collect flow's statistics on the fly. We have prototyped our architecture on the NetFPGA platform and, as an initial case study, we have implemented a module to measure packet interarrival time. This module has been validated in a realistic testbed. Our results demonstrate that the proposed architecture presents a negligible difference when compared to measurements performed by software at end-hosts.**

***Index Terms*—SDM; SDN; OpenFlow; NetFPGA**

I. Introduction

Software Defined Networks (SDNs) is an emerging network architecture that enables management of computer networks and tracking in programmable way. This paradigm consists of the separation between the control and data planes on switches [7]. In SDN, the control plane manages the network devices centrally by software and the data plane forwards packets according to the actions defined by the controller [5]. Several efforts have been made to develop SDN control applications, leaving a gap to be filled in SDN measurement and traffic tracking systems [10]. Network measurement and tracking also enables the implementation of traffic engineering techniques. However, developing these techniques for SDN networks is challenging due to the complexity and real-time constraints of these applications.

Traffic tracking plays a crucial role to enable traffic engineering in modern computer networks. Although this area was widely exploited in the past, SDN features require novel traffic engineering techniques to enable performance optimization in SDN networks [1], [2]. SDN-based traffic tracking techniques are more feasible to implement because they do not require large investments in hardware resources or network configuration. OpenFlow is an example of SDN and has emerged as a standard for communication between the controller and switches. An OpenFlow switch uses per flow counters of packet and bytes.

In this paper, we extend OpenFlow switch to implement a measurement architecture for SDN networks, enabling Software-Defined Measurements (SDM) since the controller can collect flow's statistics on the fly. We also perform measurement in a simple and scalable way without depending on end-hosts, allowing to track the performance at the granularity of flows. Our architecture has been prototyped extending the OpenFlow switch on NetFPGA 1G platform [5], [6], enabling accurate and real-time measurements of TCP flows.

As an initial case study, we have implemented a module to measure packet interarrival times. This module has been validated in a realist testbed, showing that our architecture has a negligible difference when compared to measurements performed in software at end-hosts. In fact, in our experiments, the difference between the traditional (end-host) measurements and our new proposal is close to 0%. Moreover, we present scenarios where end-host measurement is impracticable as, for example, to measure aggregated flows at a single point.

II. Prototype Implementation

In this work, we adopted the open source OpenFlow switch especially designed for the NetFPGA platform [6]. Recall that SDN paradigm migrates network complexity from forwarding network elements (switches/routers) to a centralized controller. As the centralized controller has a complete network view, it may perform fault detection and may improve network measurement tasks. Moreover, controllers allow association of flow rules inside network switches for measurement tasks, enabling to capture metrics and timestamps of the network flows using user-space programs [5].

We have modified the original OpenFlow switch [5] to gather flows mean and variance of packet interarrival time. We track packet interarrival times in flow scale, performing measurements for all packets of a flow without sampling, which scales according the number of rules supported in the switch. Currently, we track TCP flows concurrently performing measurements exactly in line-speed without affecting the packets processing rate.

Our architecture supports the description of flows in the flow entry table with the wildcard. The wildcard is useful to aggregate flows. For example, to detect if someone is doing

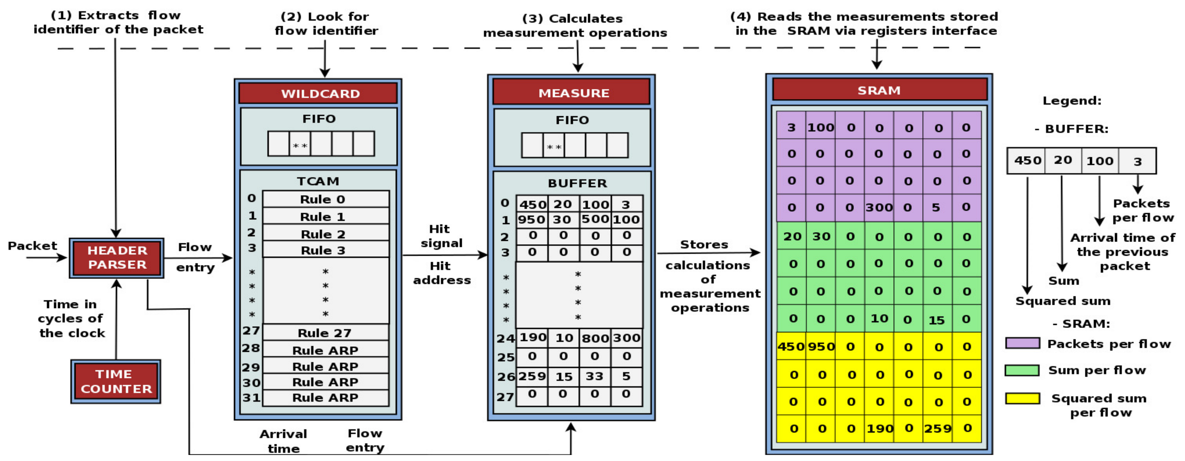

(1) Extracts flow identifier of the packet
(2) Look for flow identifier
(3) Calculates measurement operations
(4) Reads the measurements stored in the SRAM via registers interface
Packet
HEADER PARSER
Flow entry
Time in cycles of the clock
TIME COUNTER
WILDCARD
FIFO
TCAM
Rule 0
Rule 1
Rule 2
Rule 3
Rule 27
Rule ARP
Hit signal
Hit address
MEASURE
BUFFER
Stores calculations of measurement operations
Arrival time
Flow entry
SRAM
Legend:
- BUFFER:
Packets per flow
Arrival time of the previous packet
Sum
Squared sum
- SRAM:
Packets per flow
Sum per flow
Squared sum per flow

Figure 1. Datapath of the implemented NetFPGA switch.

portscan in a machine, the user could measure the packet interarrival time on her machine for any ports (wildcard port rule).

A. Data plane

Measurement tasks in the data plane begin with incoming TCP packets inside *Header Parser* module. This module extracts the flow identifier, saves the timestamp (in clock cycles) that is generated by the module *Time Counter*, and forwards this data to the *Measure* module. The *Measure* module inserts these values inside a queue. It needs to wait an output from the *Wildcard* module to decide if it drops the timestamp and flow identifier or whether performs the calculation of mean and variance. The module *Wildcard* matches the flow identifier extracted from the packet with the flow entries stored inside the TCAM. After ending the query, the module *Wildcard* sends a signal if it found the flow identifier and the TCAM address.

In next step, the *Measure* module performs calculations of mean and variance of packet interarrival time. The implementation of this metric consists of receiving the arrival time of the first packet (T_1) of the flow F_j, the arrival time of the next packet (T_2) of the flow F_j until the arrival time (T_n) of the nth packet of the flow F_j. F_j refers to the j^{th} measured flow. At each interval among packets, our prototype calculates the difference $dT = T_c - T_p$, being T_c the arrival time of the current packet and T_p the arrival time of the previous packet. Following, the summation of the difference S_j is computed, $S_j = S_j + dT$ and the summation of the squared sum of the difference, $S_j^2 = S_j^2 + dT^2$. Finally, the packet counter N_j is incremented.

After all calculations are performed, the results are temporarily stored in a buffer inside the *Measure* module and also in the SRAM. The buffer has one entry for each TCAM entry. Each buffer entry stores: the number of packet of a specific flow (N_j); the arrival time of the last packet received for that flow (T_p); the sum (S_j) and squared-sum (S_j^2) of the accumulated time intervals of the packet arrival times. When a matching occurs in the TCAM, the TCAM matched address will also address the buffer as well, where the information will be updated according the to timestamp of the current packet (T_c). The buffer works as a cache. Its main goal is to avoid the time that would be spent with repeated accesses to the SRAM. SRAM spends three clock cycles to read instructions, which may decrease the performance due to the time that the state machine waits for the data from the memory [3].

The user-space program reads, via register interface, the packet interarrival times recorded in the SRAM. Due to the limitation of logic cells number, division operations of the and variance are performed by a user-space program. In Figure 1, we show the datapath of our proposed architecture that supports measuring the mean and variance of the packet interarrival time. We highlight that our architecture is generic and allows to incorporate other metrics, which we intend for future work.

Currently, the NetFPGA 1G model, which was used in this project, supports twenty-eight TCP flows concurrently due to NetFPGA TCAM size constraints. There are other NetFPGA models with more powerful FPGAs. For comparison effects, the NetFPGA SUME [8] (FPGA Virtex-7) has 13x more resources than the model 1G, allowing to track 416 flows concurrently.

B. Control Plane

The control plane is divided in two parts: a controller and a measurement software. The controller's function is to insert the rules inside the TCAM and to inform the switch how the packet will be forwarded. There are several SDN controllers options already implemented, each one with particular characteristics. We used the POX controller. POX is implemented in Python and enables fast prototyping.

We developed an application on top of POX that insert rules containing the desired flows to be monitored on the switch. In this way, it is possible for the switch to monitor the packet interarrival time of the desired flows on the fly.

The measurement software defines the measurement process performing calculations of the mean and variance of each flow.

This software communicates with the NetFPGA through the register interface. The software reads the values of mean and variance that are stored in he NetFPGA's SRAM. For each reading, the mean and variance of the requested flows are calculated and stored in a file.

III. Evaluation

Figure 2 depicts our testbed. Our testbed is composed of two end-hosts: A and B. Host A sends Raw TCP packets at a controlled rate to host B. Our testbed has, between A and B, two traditional switches with real traffic for adding realistic delays among the packets sent by A. At the end point of the network, we inserted the NetFPGA running our prototype. Host B executes a software tool to perform packet interarrival time measurements. In B, the mean and variance were calculated after ending the experiment. In our prototype, the mean and the variance were calculated on the fly. Our goal is to compare the relative difference between the measurements in software (B) and the ones in hardware (NetFPGA) based on the predefined delays inserted among the packets sent by A.

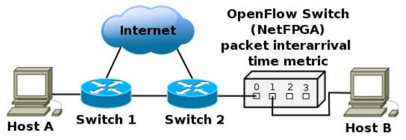


Figure 2. Testbed environment.

We developed an user-space application to send TCP packets from the host A. This application reads a file that stores the rules, the same one used by the controller. This program starts a new thread for each flow defined in the file. Each thread sends packets for a specific flow.

We defined two types of interarrival time tests: constant and random. We used constant time intervals in the validation phase for matching with the values given by our prototype. For realistic tests, we used random time intervals. The packets were generated with `Libnet` library of C language. For enabling better control of the number of packets that came from A to B and NetFPGA, we wrote the packets sent by `Libnet` in several pcap files to be transmitted by A. Each pcap file lasts one second. This approach enables performing comparisons easier and accurately. Traces format pcap were also created in the host B (software) and reproduced using DPKT library of Python language.

We performed several iterations of the experiments running the pcap files in the terminal A. Before executing them, we initialized the `tcpdump` in B for capturing packets that came from A in that moment. When ending an iteration, we read the measurements, clean both the internal buffers and the SRAM (first thirty-two lines). Then, the `tcpdump` writes an output file in B with the captured packets. These steps were performed in all iterations.

Initially, we set up five seconds for initializing the controller, the monitoring tools, and the packet transmission. Then, all experiments are performed in fifty iterations, each iteration lasting one second. Finally, we configured one second for ending these processes. Thus, the total experiment time is the summation of the initialization time, with the product of each iteration by the number of iterations, and the ending phase time: $5s + 50 \times 1s + 1s = 56s$. We discarded 10% of the samples due to the warm up period, which means 6 seconds at the beginning and ending. Each second corresponds to twenty-eight samples.

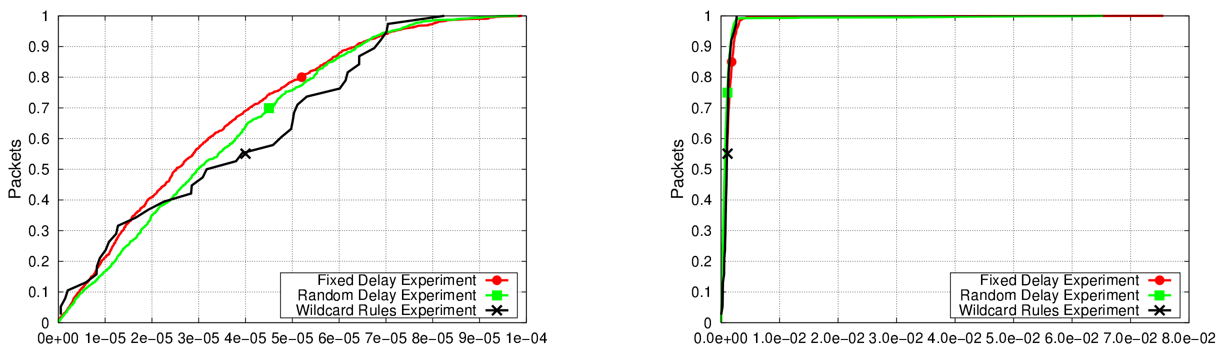
IV. Results

We designed three types of experiments. First experiment consists of predefined delays. This enables us to have the ground truth and, consequently, to validate our system. The second experiment is configured to have random delays and it represents realistic traffic pattern. Finally, the third experiment evaluates our prototype with the wildcard rules and predefined delays. Figure 3 presents mean and variance results for experiments we have performed.

Experiment 1 checks fixed delay. We send approximately 1700 packets with predefined delays of 20/40 milliseconds (ms) for each one of the 28 flows. Our results show that 95% of the relative difference between hardware and software approaches are close 0.0072% in the graph of mean. On the variance graph, 95% of the measurements obtained relative difference close to 0.26%. Three values stayed between 3% and 7%, but the absolute differences are small, presenting differences less than 10^{-4}. The expected values of the mean are inside the confidence interval $10^{-5} \times [2.897 : 3.166]$, and the expected values of the variance are inside the confidence interval $10^{-3} \times [1.0043 : 1.3574]$.

Experiment 2 tests random delay. We send approximately 1800 packets with random delays for each one of the 28 flows. We defined three seeds to generate these delays. The graph of mean showed that 95% of the measurements obtained a difference close to 0.0071%. In the graph of the variance, 95% of the measurements obtained a relative difference close to 0.19%. Also, in the graph of the variance six samples were between 3% and 6% of relative difference, but the absolute differences are small (less than 10^{-4}). The confidence interval of the graph of mean is $10^{-5} \times [3.169 : 3.432]$, and the expected values of the variance are inside the confidence interval $10^{-4} \times [7.9642 : 12.4338]$.

Experiment 3 is about wildcard rules. We sent approximately 1700 packets from host A with predefined delays 20/40ms for only four flows. We aimed to measure all these flows inside one wildcard rule. Our goal here is to show that our architecture allows measurements of aggregated flows, allowing us to track entire subnets. The graph of mean shows that 97% of the measurements obtained has a difference close to 0.007%. In the graph of the variance, 97% of the measurements had a relative difference close to 0.257%. The expected values of mean are inside the confidence interval $10^{-5} \times [2.788 : 4.349]$, and the expected values of the variance are inside the confidence interval $10^{-4} \times [8.4223 : 12.072]$.



(A) Difference measured by NetFPGA in relation to the host B

(B) Difference measured by NetFPGA in relation to the host B

Figure 3. CDF plot of the mean **(A)** and variance **(B)**.

Summary: The experiments show that the proposed architecture presents a negligible difference when compared to measurements performed by software at end-hosts.

V. Related Work

Moshref et. al. [4] discuss the tradeoff between amount of network resources and accuracy of different measurement primitives. The paper aims to evaluate measurement techniques based on counters, hashing structures and CPU programming to detect heavy hitter flows. The prototype evaluation uses different time scales and configuration in the resources allocated for each switch. They used CAIDA traces in the evaluation.

OpenSketch [10] is a Software-Defined Measurement architecture based on the SDN paradigm and uses primitives' measurement structures called sketches. Sketches are data structures that are scalable but can only provide approximation results instead of exact computation. The OpenSketch control plane is composed by one library while the data plane is separated in three stages: hashing, classification, and countering. The validation was performed by comparing the results obtained from NetFlow and the OpenSketch prototype.

OpenNetMon [9] is an open source software to perform monitoring metrics per flow, specially delay, throughput, and packet losses in OpenFlow networks. This implementation provides traffic engineering on the controller with online measurements. Packet loss is calculated with the flow statistics from the first and the last switch of each path. They subtract the packet counter from the source and destination switches. Delays and packet losses were statically defined with the Netem Linux tool.

These works illustrate that monitoring the network is important. But, none of them modified the data plane (switches) to collect new metrics beside the ones provided by OpenFlow interface, as we did.

VI. Conclusion

Traffic tracking is a key point to guarantee the quality in IP networks. Here, we designed and implemented a measurement architecture for SDN. Our architecture extends OpenFlow protocol to include other traffic's statistics. We implemented a module to measure packet interarrival time. Our architecture allows monitoring at the granularity of flows. Our system also enables software-defined measurements since the controller can collect flow's statistics on the fly.

We have prototyped our architecture on the NetFPGA platform. To validate our prototype, we compared the proposed architecture to measurements performed by software at end-hosts. Results show that the relative difference is almost zero, validating our architecture. Moreover, our architecture has better throughput performance, can collect data on the fly, and enable collecting network's statistics in the middle on the network instead of just at the end-hosts.

Acknowledgments

We thank the research agencies FAPEMIG, CAPES, CNPq and GAPSO for their financial support.

References

[1] S. Agarwal, M. Kodialam, and T. Lakshman, "Traffic engineering in software defined networks," in *INFOCOM*, 2013, pp. 2211–2219.
[2] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, "A roadmap for traffic engineering in sdn-openflow networks," *Computer Networks*, vol. 71, pp. 1–30, 2014.
[3] P. Goulart, Í. Cunha, M. A. Vieira, C. Marcondes, and R. Menotti, "Netfpga: Processamento de pacotes em hardware," *Minicursos do Simpósio Brasileiro de Redes de Computadores-SBRC 2015*, 2015.
[4] M. Moshref, M. Yu, and R. Govindan, "Resource/accuracy tradeoffs in software-defined measurement," in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013, pp. 73–78.
[5] J. Naous, D. Erickson, G. A. Covington, G. Appenzeller, and N. McKeown, "Implementing an openflow switch on the netfpga platform," in *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. ACM, 2008, pp. 1–9.
[6] NetFPGA, "Project switch openflow netfpga," https://github.com/NetFPGA/netfpga/wiki/OpenFlowNetFPGA100, nov 2015, accessed on 22/11/2015.
[7] ONF, "Software-defined networking: The new norm for networks." http://www.opennetworking.org, 2012, accessed on 23/11/2015.
[8] N. SUME, "Netfpga sume," http://netfpga.org/site/#/systems/1netfpga-sume/details/, dez 2015, accessed on: 27/12/2015.
[9] N. L. Van Adrichem, C. Doerr, F. Kuipers *et al.*, "Opennetmon: Network monitoring in openflow software-defined networks," in *NOMS*. IEEE, 2014, pp. 1–8.
[10] M. Yu, L. Jose, and R. Miao, "Software defined traffic measurement with opensketch." in *NSDI*, vol. 13, 2013, pp. 29–42.