On offloading programmable SDN controller tasks to the embedded microcontroller of stateful SDN dataplanes

Salvatore Pontarelli, Valerio Bruschi, Marco Bonola, Giuseppe Bianchi
CNIT/University of Rome Tor Vergata

***Abstract*—This paper presents a method to implement complex tasks into stateful SDN programmable dataplanes. In particular, the presented method proposes to use the internal microcontroller typically used to configure the programmable dataplane also to perform some complex operations that do not require to be executed on each packet. These operations can be executed on a set of data gathered by the dataplane and processed in a time scale that is much higher than the time window of a packet, but is much less than time scale needed for an external SDN controller. Moreover, the use of the configuration microcontroller instead of an external SDN controller to avoid the exchange of data on the control links permits a fine grain tuning of the operations to perform from the timing point of view (precise timestamping, low latency etc.). A set of measurements showing the feasibility of the method are presented, and a simple use case is described to show the effectiveness of this method to enhance the capability of stateful programmable dataplanes.**

I. INTRODUCTION

It is well acknowledged among the SDN research community that the communication between SDN controllers and SDN switches represents a bottleneck posing serious scalability inefficiencies [1],[2]. For this purpose, the SDN community effort has been focused on two research directions. On the one hand a number of works proposed to distribute multiple SDN controllers over the network topology [2],[3],[4],[5],[6]. On the other hand, several proposals started investigating the convenience of extending SDN dataplanes with stateful capabilities [7],[8],[9],[10],[11]. Indeed, besides other functional and performance advantages, extending the original stateless SDN dataplane paradigm enables the execution of simple control functions directly into the fastpath, and thus it looses the burden of the inevitable interactions with the slow path.

Nevertheless, supporting stateful operations in the dataplane does not mean being able to execute all possible network functions inside the switch. Indeed, all proposed solutions must satisfy a strict constraint on the number of clock cycles that can be allocated to process and forward the packet [12],[13] and therefore support only a limited set of (often atomic) operations, as for example read/store registers, simple arithmetic/logic operations (ADD, SUB, NOT, AND, etc..), add/remove packet headers.

Unfortunately, many fundamental Network Functions (NFs) (e.g.: Network Address Translation, Quality of Service, monitoring, firewall, etc..), demand the interaction with an external controller since they require unsupported low level primitives or data structures. Examples of such missing features include: sorting a register array, computing logarithms, exponentiation and divisions, pushing/popping to/from a stack.

978-1-5090-6008-5/17/$31.00 © 2017 IEEE

While in principle it is possible to send to an external SDN controller all the packets that must be processed to perform the required network function, this is not feasible in practice. A more viable approach consists in sampling the required flows and forwarding to the controller only a small subset of packets [15]. However, this approach requires a trade-off between the number of packets to sample and to send to the controller and the accuracy of the measurement to perform. Moreover, it cannot be applied to all the network functions because either sampling can not be applied or the inevitable latencies are not tolerable.

This approach can be further optimized by: (i) pushing the slow-path as close as possible to the fast-path (ideally on the same machine); (ii) programming the SDN controller to directly poll the data stored internally in the switch without requiring the exchange of any packet with the fastpath. However, also this optimization has a not negligible latency and data exchange overheads that make the implementation of NFs inside the fast path not scalable.

In this paper we propose a different approach to extend stateful SDN dataplane with the support of a set of "lazy" operations that will be proven to enable the executions of much more complex NFs directly in the fast path. With the term "lazy" we refer to a set of operations that are triggered by the reception of a packet inside the pipeline, but return its result after a certain amount of time without blocking the forwarding of the packet itself inside the switch.

In more details, this paper carries out the following contributions:

1) it proposes and hybrid HW/SW solution that exploits the micro-controller embedded in all SDN switch HW devices to process the "raw" data stored in the internal switch memory. In our approach, raw data represents a set of elementary measurements that are gathered in the fastpath and corresponds to the state of all the active network flows;
2) it assesses the performance of the proposed solution by providing measurements of the time required for the

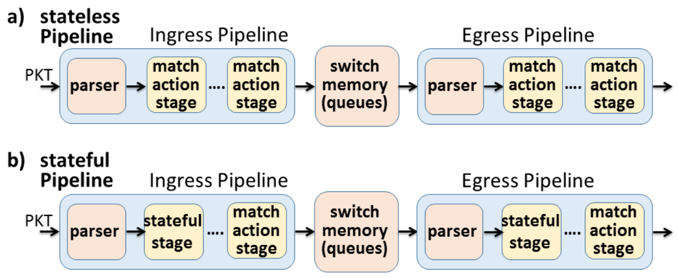

Fig. 1. a) A typical OpenFlow pipeline architecture. b) A stateful programmable pipeline architecture in which stateful stages can be pipelined with ordinary OpenFlow Match/Action stages.

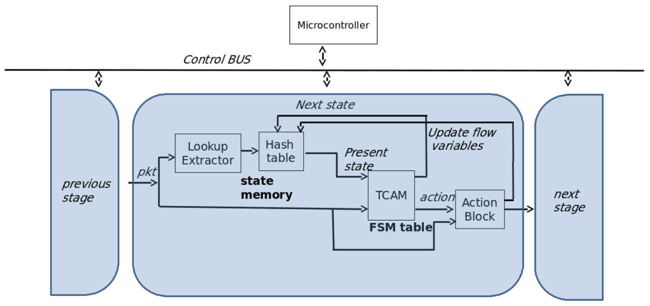

Fig. 2. a) the hardware blocks composing a stateful stage

execution of the low level operations above mentioned;
3) it shows and validates a more meaningful network function use case.

The reminder of this paper is organized as follows. Section 2 gives an overview of the reference architecture. Section 3 describes the technical details of the proposed approach. Section 4 presents and discusses the performance evaluation of our solution and section 5 concludes the paper.

II. Background

In Fig. 1 the typical Openflow pipeline architecture and our proposed stateful pipeline are shown. The stateful pipeline differs from the OpenFlow pipeline because there are some stateful stages that allow storing per-flow state in the data-path and process the incoming packets according to the value of such state. A stateful stage can implement a Finite State Machine (FSM) that selects the action to apply to a packet depending on the packet headers and on the state of the FSM. The details of a stateful stage is depicted in Fig. 2.

The stateful stage described in Fig. 1 is taken from [12], but here only the blocks that are strictly necessary to understand this work taken into consideration. The packet first goes to the look-up extractor block that build the key that is used to identify at which flow the packet under processing belongs to. After the key extraction, the hash table block provides the state information related to the selected flow. Here we consider that the state information can be composed by several flow variables [13] in which different information related to the flow can be stored (e.g. number of received packets, timestamp of the last packet, number of bytes of the packet etc.). Then, the current state and the packet header are passed to the TCAM table that selects which actions must be executed, and also decides how to update the flow state (next state). The selected actions are passed to the action block to be actually applied, while the update state information is given back to the hash table with a feedback loop. The action blocks can also update the value of the flow variables (e.g. updating the per-flow counters) stored in the state memory.

The prototype that implements the stateful pipeline [12] hosts a microcontroller that is used to configure all the blocks composing the pipeline. In particular, the microcontroller can access all the memories (configuration registers, TCAM and hash tables) and configure the lookup extractor to select the header fields needed to build the flow-key. In order to configure the elements composing the stateful pipeline, each component is memory mapped in the address space of the microcontroller, which can directly read/write the content of these components. This direct interaction between the microcontroller and the prototype core can be further exploited to perform some *house-keeping operations*, like for example aging the entries stored in the state table. In fact, it is possible to program the microcrontroller to perform a periodic scrubbing of the state table, removing all the entries older than a specific threshold. For this purpose, two activity flag bits are stored in each flow entry and permit to label entries as ACTIVE, INACTIVE (no accesses have occurred in a configurable management cycle, e.g., order of seconds), and DELETED. When a management cycle is initiated, the microcontroller changes the status of the ACTIVE entries to INACTIVE, and of the INACTIVE entries to DELETED. The data-path instead moves an INACTIVE entry to ACTIVE when the state is read/write and uses the DELETED entries as empty slot to store the state of new incoming flows. Finally, the microcontroller is used to communicate with an external SDN controller by using a suitable interface (a PCI interface if the system is hosted in a PC or an Ethernet port for a stand-alone system).

III. Proposed approach

The proposed approach splits the computation of complex operations in two steps: the first step is executed at packet speed and gathers some intermediate flow information that are stored in the state memory; the second step periodically scans the state memory and elaborates the intermediate results to provide the final output of the complex operation.

Possible examples of this approach are the computation of the average packet length and the detection of the heavy hitter flows. The computation of the average packet length of a flow requires to store the number of packets and the number of bytes transmitted in a time interval in the state memory. This information is stored in the flow table and is updated each time a new packet belonging to a specific flow arrives. Updating the number of packets and the number of bytes only requires that the pipeline logic is able to perform simple arithmetic operations such as increment and additions. Computing the average requires to perform a division between the two per-flow variables stored in the state memory. In our approach, this operation is performed by the microcontroller with a specific period, so that the average length of the packet is computed and stored in a flow variable. Since the microcontroller write the computed average value directly in the state memory, this value can be directly used by the dataplane to perform specific network functions (e.g. forwarding a flow based on the packet length). Another possible use case is the detection of the N heavy hitters. Here the raw information is the number of packets of each flow, and the microcontroller task is programmed to scan the state memory and retrieve the maximum N values among the number of packets of each flow. Then, the microcontroller can update the state memory and label these N flows as heavy hitters. We remark that this is an approximate method, since new packets can arrive during the scanning period in which the microcontroller is used to search the maximum, and is "lazy" in the sense that the detection of a heavy hitter arrives with a certain delay, that correspond to the frequency at which the microcontroller performs the maximum search task. the proposed method can be refined if during the scanning operation the microcontroller also counts the overall number of incoming packets as the sum of the packets counted for each flow. This sum corresponds to the incoming rate expressed in terms of pkts/sec. This metric allows a better identification of heavy hitters, since we can selects as heavy hitters only the flows that are above a certain threshold of the incoming rate. The time needed by the microcontroller to compute the intermediate results depends on several factors such as the complexity of the operation to perform, the hardware processing capability of the microcontroller and amount of data to process. For this last factor, we can configure the microcontroller in two modes.

In the first mode, the microcontroller fetches all the rows of the state memory and internally discards the rows that are INACTIVE or DELETED. In the second mode the microcontroller first checks if the row is ACTIVE, and only in this case retrieves the entire row storing the flow variables related to the ACTIVE flow. In the first mode, which we will call non-controlled (NC), the microcontroller will require the same number of clock cycles to perform the required operation mainly independently from the number of active flows in the state memory. In the second mode, which we will call controlled mode (C), the number of clock cycles will depend on the number of active flows.

The approach that provides the highest processing throughput depends on the number of flow variables to fetch for each active flow and on the number of active flows. Finally, the difference in processing throughput between the two approaches is only significant when the complexity of the operation to be executed inside the microcontroller is comparable with the time needed by the microcontroller to read (and write-back if needed) the flow variables. In the next section the dependency of the processing throughput on the operation complexity and on the number of active flows with the two above described modes is reported. The measurements has been carried out using the [13] prototype developed on top of the last generation NetFPGA [16].

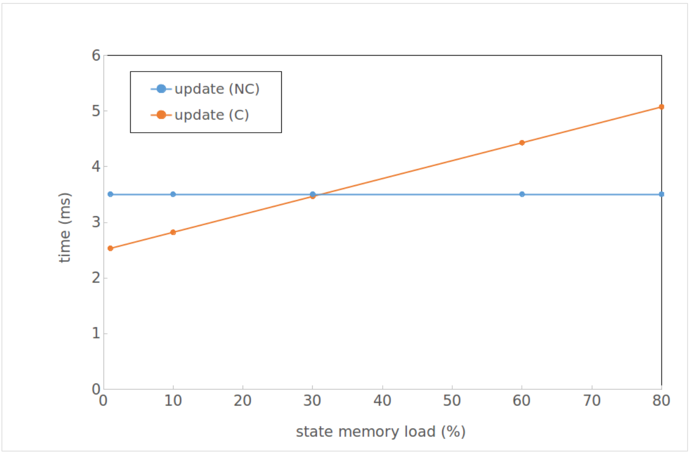

Fig. 3. time required (in ms) to scan the state memory with the update operations. The controlled (C) and non-controlled (NC) modes are reported.

IV. Measurements

The first set of measurements have been performed to assess the performance of the basic operations on the state memory. In particular, we tested the table context update, using a sequence of read/write operations. The experiments have been carried out using a 64KB memory divided in 2K row of 32 bytes with the table occupancy (i.e. the number of active flows) varying from 10% to 80%[1]. Each row can be individually set as ACTIVE/INACTIVE/DELETED. The operations have been tested with the two modes and the results are reported in fig. 3. The x-axis reports the percentage of table occupancy, while the y-axis reports the time require to complete the table scan.

As can be seen, the controlled mode requires less time when the percentage of table occupancy is less than 30%, while is worst than the uncontrolled mode for table occupancy greater than 30%. It is important to note that the difference between the two modes is significant, since it reaches the 44% in the worst case (table occupancy of 80%). It should be noted that this experiment provides an estimation of the minimum overhead required by our approach, since the actual computing time of the microcontroller is negligible.

A second set of measurements is reported in fig. 4. This experiment takes into account the computation of the average packet length using as intermediate values the overall number of bytes and the number of packets of each flow. The average is computed by the microcontroller using a division operation between the intermediate results. As mentioned before, the time required to complete this operation also depends on the characteristics of the microcontroller. In this experiment, we used two microcontroller configurations: (i) a low resource microcontroller with a dedicated hardware multiplier that is exploited to speed-up the division and (ii) a microcontroller

[1]we do not test higher loads to avoid issues related to the hash table insertion algorithm.

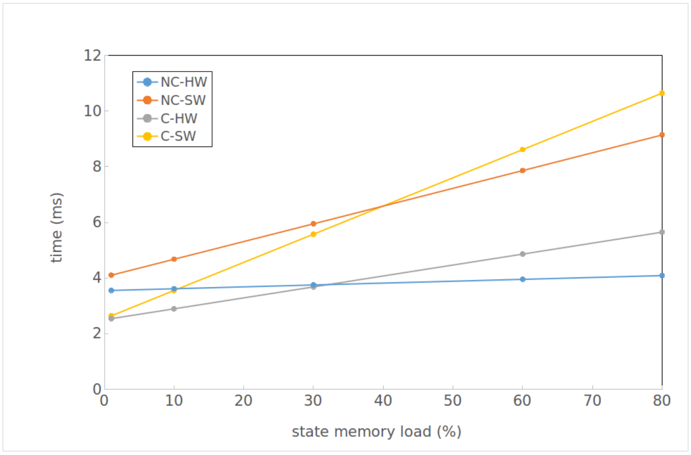

Fig. 4. time required (in ms) to perform an average operation: the controlled (C) and non-controlled (NC) modes are reported when microcontroller is equipped with a hardware multiplier (HW) or without it (SW).

without a dedicated hardware multiplier in which the division is purely software. These two configurations permit to identify the impact of the microcontroller characteristics on the overall performance achievable by our proposed method.

For the second experiment the controlled mode gives better performance when the table occupancy is small. Instead, the for higher load the NC mode should be preferred. However, the difference between the two modes is less significant when the overall time required to perform the operation grows. In fact, when the purely software division is used, the difference between the NC and the C mode is less than 20%. The figure also shows the different performances achievable when the HW multiplier is exploited. The time required to compute the average on the whole table is less than 50% compared with the time required when the purely software division is used. The number of clock cycles required to perform a purely software division is not negligible, because of the following two factors: (i) the time required to move the data between the state memory and the internal microcontroller registers and (i) the time required to actually perform the division. Instead, when the hardware multiplier is exploited the actual computing time of the microcontroller is negligible and then the overall time required is almost the same of the first experiment.

V. Conclusions and Future Work

In this paper we proposed a method to implement complex tasks into a stateful SDN programmable dataplane using a "lazy" approach. We identified a set of functions that do not require to block the process of the packet in the pipeline to be executed and can be delegated to a microcontroller that is connected to the dataplane with an internal data bus. In particular, we propose to split the actual computation in two phases: a first phase is executed in the dataplane and is used to gather some intermediate information and a second phase that is used to elaborate the intermediate data. We presented a set of measurement to assess the performance of our method and how these performance depends on the complexity of the application and on the microcontroller processing performance. Finally, we identified a set of useful network function that can be efficiently realized using our approach. The method proposed in the paper could be widely applied, since almost all programmable dataplanes already have an internal microcontroller used to configure the memory tables and to provide debug capabilities. Until now, these features are not exposed to the user, due to the lack of a standardized API that allows the external SDN controller to program the internal microcontroller. We believe that the definition of this API is a mandatory tool to ease the implementation of the method described in this paper.

VI. Acknowledgment

This work has been partly funded by the EU commission in the context of the BEBA project, Grant Agreement no. 644122 and SUPERFLUIDITY project, Grant Agreement no. 671566.

References

[1] S. H. Yeganeh, A. Tootoonchian and Y. Ganjali, "On scalability of software-defined networking," in IEEE Communications Magazine, vol. 51, no. 2, pp. 136-141, February 2013.

[2] K. Phemius, M. Bouet and J. Leguay, "DISCO: Distributed SDN controllers in a multi-domain environment," 2014 IEEE Network Operations and Management Symposium (NOMS), Krakow, 2014, pp. 1-2.

[3] A. Dixit, F. Hao, S. Mukherjee, T. V. Lakshman and R. R. Kompella, "ElastiCon; an elastic distributed SDN controller," 2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS), Marina del Rey, CA, USA, 2014, pp. 17-27.

[4] S. Sezer et al., "Are we ready for SDN? Implementation challenges for software-defined networks," in IEEE Communications Magazine, vol. 51, no. 7, pp. 36-43, July 2013.

[5] T. Koponen et al. "Onix: A distributed control platform for large-scale production networks," in proc. of Operating Systems Design and Implementation (OSDI), Vol. 10. 2010.

[6] P. Berde et al. "ONOS: towards an open, distributed SDN OS," in proceedings of the third workshop on Hot topics in software defined networking. ACM, 2014.

[7] G. Bianchi, M. Bonola, A. Capone, C. Cascone, "OpenState: programming platform-independent stateful openflow applications inside the switch", ACM SIGCOMM Computer Communication Review, 44(2), 44-51.

[8] M. Moshref, et al. "Flow-level state transition as a new switch primitive for SDN", Proceedings of the third workshop on Hot topics in software defined networking. ACM, 2014.

[9] A. Sivaraman et al. "Packet transactions: High-level programming for line-rate switches", Proceedings of the 2016 conference on ACM SIGCOMM 2016 Conference. ACM, 2016.

[10] P. Bosshart et al. "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," ACM SIGCOMM Computer Communication Review. Vol. 43. No. 4. ACM, 2013.

[11] P. Bosshart et al. "P4: Programming protocol-independent packet processors," ACM SIGCOMM Computer Communication Review Vol. 44. No. 3. ACM, 2014.

[12] S. Pontarelli, M. Bonola, G. Bianchi, A. Capone, C. Cascone, "Stateful Openflow: Hardware Proof of Concept", IEEE HPSR 2015.

[13] G. Bianchi et al, "Open Packet Processor: a programmable architecture for wire speed platform-independent stateful in-network processing", on line available at https://arxiv.org/abs/1605.01977

[14] J. Matias, J. Garay, N. Toledo, J. Unzilla, and E. Jacob. "Toward an SDN-enabled NFV architecture," IEEE Communications Magazine 53, no. 4 (2015): 187-193.

[15] S. Shirali-Shahreza, G. Yashar "FleXam: flexible sampling extension for monitoring and security applications in openflow," Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking. ACM, 2013.

[16] N. Zilberman et al. "NetFPGA SUME: Toward 100 Gbps as research commodity," IEEE Micro 34.5 (2014): 32-41.