A Novel Pflua-based OpenFlow Implementation for VOSYSwitch

Jeremy Fanguede, Michele Paolino, Dimitar Dimitrov, and Daniel Raho Virtual Open Systems
Grenoble, France
Email: {j.fanguede, m.paolino, dimitar, s.raho @virtualopensystems.com

***Abstract*—Software Defined Network (SDN) along with Network Function Virtualization (NFV) paradigms are emerging as a viable alternative to traditional network architecture. The Fog and Mobile Edge computing network and the Internet of Things infrastructure are adopting these concepts as well. In this context, virtual and SDN-enabled switches are crucial. This paper presents the novel OpenFlow software implementation of VOSYSwitch, the high performance virtual switch solution developed by Virtual Open Systems. Written in Lua, such OpenFlow implementation takes advantage of the library *pflua* and of the LuaJIT just-in-time trace compiler to translate OpenFlow flows into Lua code via the PCAP Filter language. As a consequence, the code executed to perform the flow matching and actions is generated on demand and tailored to the flow entries of the forwarding tables of the dataplane. Benchmarking activities, whose results are included in this paper, have been performed to compare this new OpenFlow switch implementation with the de-facto standard OpenFlow switch solution OVS-DPDK. The results show that VOSYSwitch OpenFlow implementation is superior to OVS-DPDK for environments with simple flow configurations and low number of entries, typical of Edge Computing.**

***Keywords*—Software Switch, OpenFlow, pflua, LuaJIT, software switch benchmark, VOSYSwitch, OVS-DPDK.**

I. Introduction

Virtual switch solutions with configurable data-plane are one of the key component of Software Defined Networking (SDN) [1] and Network Function Virtualization (NFV) [2]. These concepts are also emerging in Edge and Fog Computing [3][4] and Internet of Things (IoT) applications, where they bring a much more flexible network architecture [5]. One of the most common way to control an SDN-enabled switch is the OpenFlow [6] protocol, which enables the possibility to add, update, remove and monitor the switch forwarding flow tables and their flow rules. Thus, OpenFlow-enabled virtual switches are becoming central elements of modern network architecture, including edge infrastructures.

In this paper we present the novel OpenFlow implementation added to *VOSYSwitch* [7], the Virtual Open Systems virtual switch solution based on the open-source framework *Snabb* [8][9]. Such OpenFlow implementation (based on specification version v1.3 [10]) has the distinctive particularity to translate the OpenFlow flow entries dynamically in Lua language via the PCAP Filter language [11] (also used in the popular *tcpdump* tool). For such translation, it leverages on the fast Lua network packet filtering library *pflua* [12], which is able to compile PCAP filters, and on the *just-in-time* trace compiler LuaJIT [13]. It is worth mentioning, that this translation concept can be applied to other network standards equivalent to OpenFlow, such as P4 [14]. Additionally a set of benchmark is included to showcase the performance reached by this OpenFlow implementation.

The remaining part of the paper is organized as follows: Related works are presented in Section II and in Section III the VOSYSwitch OpenFlow bridge architecture and implementation is described. Section IV provides performance results and analysis while Section V concludes the paper.

II. Related Work

Software switches are firstly used to provide packet switching between virtual machines, but nowadays they can also be used to turn a dedicated machine into a smart switch, as their performance is raising up.

Several virtual switches with support for OpenFlow have been implemented. For instance, Open vSwitch (OVS) [15] is an OpenFlow software switch widely used, designed for hyper-visors platform. Pfaff et al. [16] described its design and implementation in 2015. It consists of two main components, an userspace daemon and a datapath kernel module which is written for each supported hypervisor/host. The flow classification mechanism and the caching techniques that OVS uses are quite advanced and evolved over time. In addition to the kernel datapath, OVS also implements a datapath in userspace, this implementation plays its full role when used in combination with userspace drivers. A version of OVS based on the Intel's Data Plane Development Kit (DPDK) [17] framework, OVS-DPDK, takes this approach and shows significant performance improvements compared to OVS in-kernel module [18]. The performance improvements are also resulting from the usage of DPDK user-space polling mode drivers for the network cards. Emmerich et al. [19] reported a forwarding rate (for a OpenFlow bridge configured with a single flow) of 11.31 Mpps for a DPDK enabled version of OVS for the smallest packet size on a Intel Xeon E3-1230 V2 CPU.

Rahimi et al. [20] proposed, *Lagopus*, a software OpenFlow switch leveraging also on the Intel's DPDK drivers and libraries. It is specifically designed to be run in multi-core processors. For instance, a forwarding rate of 9.8 Gbps is reported for 1500 bytes packets with an OpenFlow table

978-1-5386-5896-3/18/$31.00 ©2018 IEEE

containing 100K entries matching on MPLS label on a server with a Intel Xeon E2660 v3 CPU.

Benchmarking tools for OpenFlow are generally focused on the controller side [21] more than on the datapath side. Although, some tools that allow datapath tests also exist, such as OFLOPS [22] which is a software framework to develop tests for OpenFlow-enabled switches. But, this software framework does not support the most recent versions of OpenFlow such as the v1.3 implemented in the switch described in this paper.

III. VOSYSwitch and its OpenFlow implementation

This section presents background information about the Virtual Open Systems virtual switch solution VOSYSwitch and the *pflua* library respectively in sections III-A and III-B. After, VOSYSwitch OpenFlow implementation architecture is described in section III-C.

A. The VOSYSwitch virtual switch

VOSYSwitch is based on the Snabb framework [8], which uses the Lua programming language to build fast network applications. It leverages on the LuaJIT trace compiler to "compile" the Lua code and allow self-adaptable runtime code generation, resulting in dynamic code path optimizations depending on the processed data [9]. The network applications are designed as a packet processing graph. Each node of the graph is a one-purpose network component, which can be, for instance, a driver for a network card, a firewall or a learning bridge. VOSYSwitch enhances the implementation of these concepts to create a NFV-ready software switch. The processing graph, defining the components and their links, is described thanks to a JSON file that can be written directly by the network administrator or via the OpenStack networking service Neutron [23]. Furthermore, it implements a master-workers multi-thread paradigm based on memory communication that allows processing graphs to be executed by multiple CPU cores. One or more components of a processing graph can be assigned to a specific CPU core (Figure 1). The VOSYSwitch and Snabb architecture and their performance has been described in detail by the authors in a previous work [7]. The OpenFlow extension of VOSYSwitch actually consist of a new component, the OpenFlow bridge, that can be used as a node in the processing graph. It works as a bridge and can be connected to other components of Snabb/VOSYSwitch (Figure 1). Thus, it may be used in combination with others components such as tunneling nodes (e.g., VxLAN, GRE, etc.).

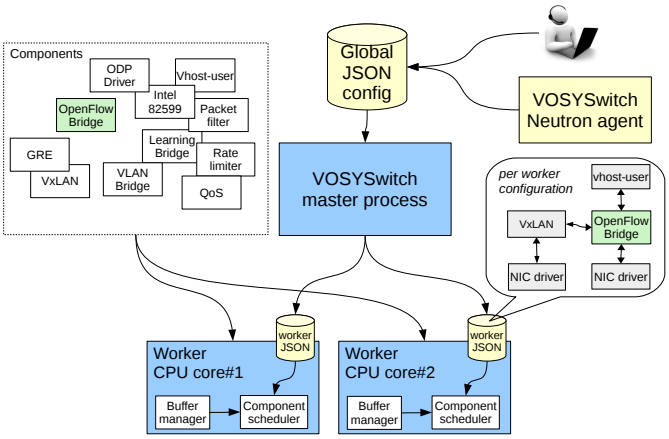


Fig. 1. VOSYSwitch architecture

B. PCAP Filter Language and pflua

The OpenFlow bridge implementation presented in this paper takes advantage of the "PCAP Filter" language to translate OpenFlow flow rules into Lua code. More in particular, "PCAP Filter" (hereinafter referred as *PF filter*) is a language with a quite powerful and complete syntax [11], developed for the widespread network traffic dump tool *tcpdump*. In *tcpdump* the parsing and compilation operations are performed by the PCAP library *libpcap*, which compile the PF filter into a low-level byte code that can be interpreted, by the same library to filter packets over a network interface.

pflua [12] is the alternative to *libpcap*, developed by Igalia, which is able to compile the PF filters into Lua language. It has been developed specifically to create filtering applications inside the Snabb framework and provides better performance than *libpcap* [24] by executing the generated Lua code with the LuaJIT just-in-time compiler.

In this context, the authors of *pflua* also built *pfmatch* [12], a language which leverages on *pflua* to set multiple PF filters conditions and associate them to specific actions (i.e., Lua functions). More in particular, a *pfmatch* program is composed of a ordered list of *Clause* items, each containing a condition (*Test*) which is a PF filter, associated with a *Dispatch* action. Figure 2 shows an example of the *pfmatch* syntax, where a custom *Dispatch* function (i.e., `handle_arp()`, `forward()`, `drop()`, etc.) is associated to a *Test* condition. Those functions when compiled into Lua will be called with at least two parameters; the packet pointer and the packet length plus some optional arguments such as the pointers calculated by the compiler (e.g., the IP header position).

```
match {
  -- 'Test' => 'Dispatch'
  arp  => handle_arp()
  ip src 192.168.1.0/24 => forward(2)
  ip dst 10.0.1.2 => change_src(&ip[0])
  => drop()
}
```

Fig. 2. pfmatch code example, composed of four filters (i.e., *Clause*)

pfmatch is particularly useful to dispatch network packet according to their headers. In fact, its dispatching mechanism is interesting because it can fit the concept of a network datapath such as the one of OpenFlow. In the case of OpenFlow [10], every *Clause* can be mapped to a OpenFlow *Flow Entry*, the *Test* expression to the *Matching fields* and the *Dispatch* function to the *Instruction Set*.

C. VOSYSwitch OpenFlow bridge architecture

The VOSYSwitch OpenFlow bridge uses the *pfmatch* language to create optimized dispatching code tailored to the flow entries of an OpenFlow table. For that purpose, the *pfmatch* code is re-generated to be adapted to the new flow entries of the table during the switch initialization or when a flow table is modified. The different translation steps to generate the code for an OpenFlow flow table are summed up in Figure 3.

Two sets of Lua code (③ and ④ in Figure 3) are generated during the translation flow. The first one, ③, is resulting from the compilation of the *pfmatch* code by *pflua*, it contains a single Lua function performing the filtering and calling functions contained in the second set: ④. The latter, are functions performing the actions of the instruction set of the flow entries, there is one function per flow entry. Previously, the flow table defined using the OpenFlow Protocol (OFP) [10] (①) has been translated into *pfmatch* language (②) as described in Section III-B. Finally, the obtained Lua codes are run as a normal Lua code using LuaJIT (⑤).

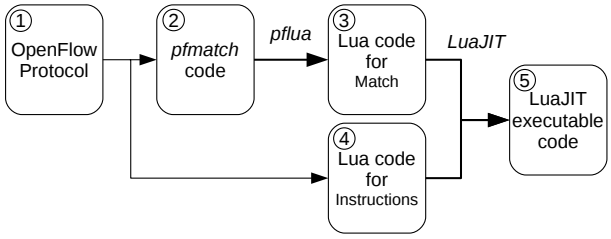


Fig. 3. pflua based flow table translation steps

To better explain this translation, a simple example of a single OpenFlow table with three flow entries can be taken as an example. The first flow entry, `(1)`, which has the highest priority, matches on the source MAC address and drop the packet, the OFP flow match field type `OFPXMT_OFB_ETH_SRC` [10] is used with `F6:D7:EF:15:13:28` as value, no instruction set is provided in order to drop the packet [10]. The second, `(2)`, matches TCP packets on a range of IPv4 destination addresses: 192.168.0.0/16, for this it uses three OFP flow match fields, `OFPXMT_OFB_ETH_TYPE` to match $0x800$ (the IPv4 ethertype), `OFPXMT_OFB_IP_PROTO` to match on TCP protocol number, 6, and `OFPXMT_OFB_IPV4_DST` with a mask option to match on 192.168.0.0/16 IPv4 range, its instruction set consist, for instance, of a single action to output on a port. The third entry, `(3)`, has the lowest priority, and it is used as a "table miss" entry, it does not have any fields, meaning it match all the packets, its instruction set is empty to drop the packet, for example. The resulting *pfmatch* generated by the switch of such a OpenFlow table is shown in Figure 4.

```
match {
 ether src F6:D7:EF:15:13:28 => ins1()                    -- (1)
 ether proto 0x800 and proto 6 and (ip and
     dst net 192.168.0.0 mask 255.255.0.0) => ins2() -- (2)
 => ins3()                                                -- (3)
}
```

Fig. 4. pfmatch generated code example of an OpenFlow flow table

The OFP flow match fields are translated into PF filters in a quite straightforward way, there is almost a PF filter equivalent for each of the OFP flow match header fields. Even if there is some replications in the filters like for flow entry `(2)` where `ether proto 0x800` and `ip` filters are actually equivalent, *pflua* simplifies the filters at compilation stage. The generated Lua code compiled by *pflua* for this example is shown in Figure 5. The handling of the different flow entries

```
local band = require("bit").band
local cast = require("ffi").cast
return function(self,P,length)
   if length < 12 then goto L5 end
   do
      if cast("uint16_t*", P+6)[0] ~= 55286 then goto L5 end
      if cast("uint32_t*", P+8)[0] == 672339439
      then
         return self:ins1(P, length)
      end
      goto L5
   end
::L5::
   if length < 34 then goto L11 end
   do
      if cast("uint16_t*", P+12)[0] ~= 8 then goto L11 end
      if P[23] ~= 6 then goto L11 end
      if band(cast("uint32_t*", P+30)[0],65535) == 43200
      then
         return self:ins2(P, length)
      end
      goto L11
   end
::L11::
   return self:ins3(P, length)
end
```

Fig. 5. Generated Lua code example of an OpenFlow table (`P` is the address of the packet data, `length` is the length of the packet).

can be easily recognized in this function. Calling this function with a network packet as argument (pointer on packet data and packet length) will actually call the *instruction set* function of the flow entry matched, (`ins1()`, `ins2()` or `ins3()` in this case). The instruction functions are generated separately by the switch according to the instruction set of the flow entries of the table, for instance a pseudo code for an output action to port number 2 can be simply: `transmit_to(P, length, 2)`.

Such generation of LuaJIT executable code from OFP commands provides a way to dynamically optimize the execution of the forwarding actions. However, the standard implementation of *pflua* today supports only matching on the data of the packet, because it only implements the PF filter language. This is an important limitation because OpenFlow defines, in addition to the flow match on header fields, also flow match on pipeline fields. Those fields (e.g., ingress port number, metadata and tunnel ID) are value attached to packets, not included in the header fields. As a consequence, those pipeline fields cannot be matched with a PF filter since it only supports matching on the data of the packet.

In order to overcome such limitation and to support the metadata and tunnel ID fields in VOSYSwitch, *pflua* has been extended to be able to match on the additional packet fields provided by the packet structure of the switch. In addition to the packet data pointer and length, the parameters of the generated function has been extended to also use a custom variable that can be the specific packet structure of the switch. However, for performance reasons, this *pflua* extension is not used to support the ingress port matching (mainly OFP field

OFPXMT_OFB_IN_PORT), in order to avoid an additional branch condition in the critical code path. Instead, it has been chosen to create an instance of the OpenFlow flow table chain for each ingress port. That is to say, if a flow entry of a OpenFlow table contains an match for OFPXMT_OFB_IN_PORT, the matching will only be performed in the flow table variant belonging to this ingress port. Thus, there is no condition check on the ingress port number in the datapath, meaning that there is no performance penalty of using the ingress port field matching.

Moreover, the implementation being written in Lua the full OpenFlow datapath implementation (version v1.3) is quite concise and fit in 3K lines of code. While most of the optional Openflow v1.3 features have been implemented, such as all the match fields (including metadata and tunnel ID), the set-field actions and the *Group* feature.

IV. Performance Benchmark

In order to evaluate this novel OpenFlow implementation, this paper presents a performance benchmark of the developed solution against OVS-DPDK, the forwarding rate of the datapath implementation with different set of flow entries is measured. Figure 6 shows the test scenario set up to perform the benchmark measurements, this scenario allows to stress an OpenFlow bridge by saturating its interfaces. The

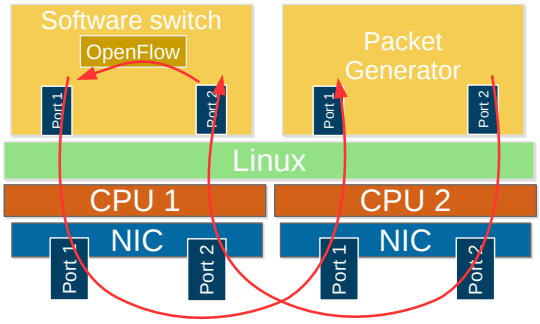


Fig. 6. Benchmark environment.

software switch solution under test (i.e. VOSYSwitch with its OpenFlow bridge or OVS-DPDK) and the software traffic generator *MoonGen* (an open source packet generator built on top of DPDK/LuaJIT and able to saturate a 10 Gbit/s Ethernet link [25]) run isolated on dedicated socket. Moreover, two dual port 10 Gbit/s network PCI cards are used, one for the software switch, and one for the generator. The two cards are connected back to back (through a 10Gbit/s hardware switch). The software switches are tested with a single OpenFlow bridge with two ports, each port is connected to a physical port of the associated network card. The drivers used for the network card are the DPDK polling mode driver in the case of OVS-DPDK and *MoonGen* while VOSYSwitch uses the *Snabb* driver.

The benchmark are run on a server with two Intel Xeon Processors E5-2630 v3 (Haswell architecture, 8 cores) at 2.4GHz, on which Hyper-Threading, *Turbo Boost* mode, and frequency throttling have been disabled in order to get reproducible results. The memory consist of 64GB of DDR4 RAM (at 2133MHz), both software switches tested (OVS-DPDK and VOSYSwitch) and the *MoonGen* generator are configured to use 2MB huge-pages. The machine is running a Debian-based Linux distribution, the OVS-DPDK version benchmarked is *v2.8.0*, while DPDK version is *v17.05.1*. The network card used by the software switches is an Intel 82599ES while the one used by the software generator is an Intel X710. The switch solutions are statically assigned to dedicated cores (isolated by Linux) in the same NUMA node as the network card.

Three OpenFlow configurations have been tested, with multiple numbers of flow entries. The first one measures the *Forwarding Performance* (see results in Section IV-A), and therefore, match only on the port number, i.e, a single rule with OFPXMT_OFB_IN_PORT as match field is inserted. The second configuration is matching on the port number and on the destination MAC address (OFPXMT_OFB_ETH_DST), it measures the *L2 Forwarding performance* (results in Section IV-B). The last one measures the *L3 Forwarding performance* (results in Section IV-C) and match on the port number and on the destination IP address (OFPXMT_OFB_IPV4_DST plus the needed prerequisites [10]).

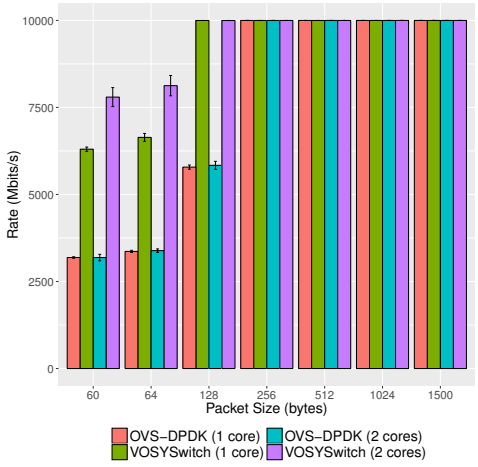
The generator sends packets according to the configuration tested, i.e., if 100 flow entries matching on IP destinations are used, then the generator sends 100 packets matching each entry of the table, in a loop. The number of flow entries tested is, on purpose, limited to low values, to be inline with a generic fog or edge compute node which does not have a high number of flow entries. Seven packets sizes are tested from 60 bytes up to 1500 bytes. For each measurement iteration of the test, the generator measures the throughput of packet received, thus, the theoretical maximum is 10 Gbits/s. The average value of the received throughput is presented in the following graphs along with an error bar representing the 95 percent confidence interval of the measured samples.

A. Forwarding Performance

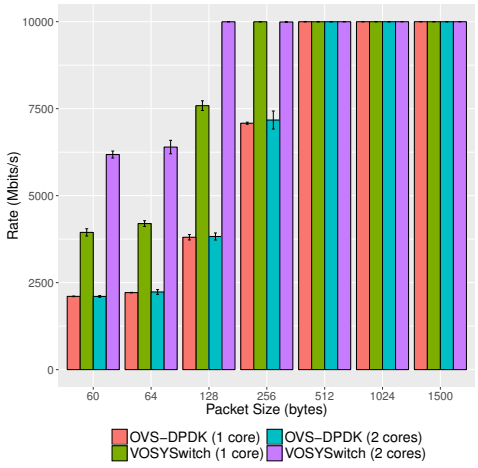
A simple OpenFlow configuration with a single table containing a single flow entry that simply forward all the packets from one port to the other is used, the only match field inserted is: (OFPXMT_OFB_IN_PORT). The instruction set consist of a single *Apply-Action* [10] with an output action (OFPAT_OUTPUT). Each switch tested run on a single CPU core, the goal of this benchmark is to stress the switch datapath in the most simple case. The results are similar for both switch solution, for packet sizes higher or equal to 128 bytes, they reach the line rate (i.e., 10 Gbit/s). For a packet size of 64 bytes, the forwarding rate is similar: OVS-DPDK reach 9.85 Gbit/s and VOSYSwitch 9.86 Gbit/s.

B. L2 Forwarding Performance

These scenarios aim to test the OpenFlow switch when it behaves similarly as a L2 learning bridge, i.e., the flow table is filled with flow rules matching the L2 header (MAC addresses)



(a)



(b)

Fig. 7. Forwarding rate with 10 (a) and 100 (b) flow entries matching on L2 header

associated with an action transmitting to the right port. The bridge of both solutions have been configured with different number of rules matching on the destination MAC address and transmitting packets to the other port. The generator sends packets with the same number of MAC addresses than the number of entries in the flow table. Figure 7a shows the result for the case where 10 L2 rules are inserted. Line rate is achieved by VOSYSwitch starting from packet size of 128 bytes, while it is only at 256 bytes for OVS-DPDK. Therefore the throughput achieved, starting from these packet sizes, is the maximum reachable by the hardware: 10 Gbit/s. For smaller packet sizes VOSYSwitch clearly gives a better throughput. For instance, when a single core is used, for a packet size of 64 bytes VOSYSwitch reach a throughput of 6.6 Gbits/s while OVS-DPDK only achieve 3.4 Gbits/s. When two cores are used, VOSYSwitch gives even better results: 8.0 Gbits/s for 64 bytes packets. OVS-DPDK does not show any improvement when using two worker threads instead of one, this is due to the fact that OVS allocates one worker thread per

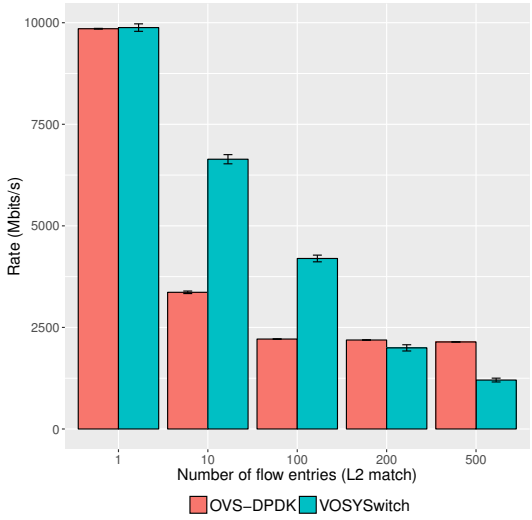
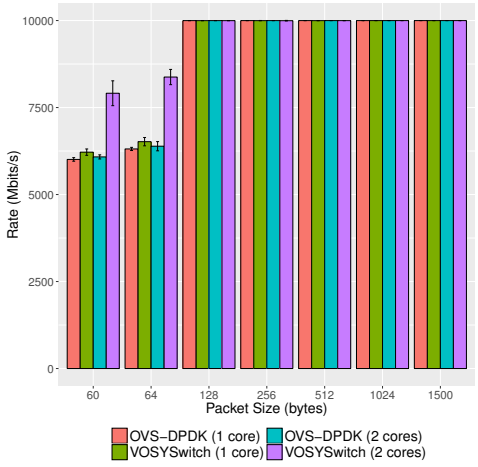


Fig. 8. Forwarding rate of 64 bytes packet according to the number of flow entries (matching on MAC addresses)

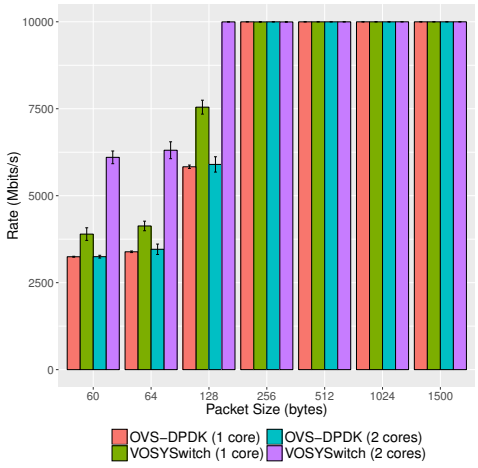
ingress port, while VOSYSwitch when two cores are available, uses one worker thread to handle the two ports and another one to handle the OpenFlow datapath. Therefore with this configuration, in OVS only one thread is really contributing to the throughput, thus limiting the scalability of the solution in this particular case. The results for the same configuration but when 100 flow entries are inserted are shown in Figure 7b. For VOSYSwitch, as for the 10 flow configuration, the line rate is achieved with a packet size of 256 bytes (still 128 bytes if 2 cores are used). Smaller packet sizes exhibit a lower throughput, for instance for 64 bytes packet on a single core, the rate is 4.2 Gbits/s (was 6.6 for 10 flow entries). OVS-DPDK, however, only reach the line rate with packet size of 512 bytes. For smaller packet size the rate is lower than with the 10 flow configuration, for instance it is only 2.2 Gbits/s for 64 bytes packets (versus 3.4 for the 10 flow configuration). Overall, for the L2 forwarding rate, for the numbers of flow tested, VOSYSwitch highlights significant better performance. But with an higher number of flow entries VOSYSwitch starts to have its throughput degraded as exhibited in Figure 8. This is due to the fact that the code used to perform the matching is growing linearly, following the number of flows. This is a challenging problem that need to be addressed in the future, for example by adding a cache mechanism. Instead, OVS-DPDK, after 200 flow entries have an almost steady rate, this is the strength of OVS-DPDK.

C. L3 Forwarding Performance

Similarly to the L2 forwarding performance test, the OpenFlow bridge as been configured with flow entries matching on the L3 header. The bridge acts as a router, the flow entries match on the IP destination address and output the packet to the other port. Both switch solutions have been tested with 10 and 100 flow entries. The results of VOSYSwitch is quite similar than for the L2 performance rate as depicted



(a)



(b)

Fig. 9. Forwarding rate with 10 (a) and 100 (b) flow entries matching on L3 header

in Figure 9a, for a packet size of 64 bytes (single core) it achieved a rate of 6.5 Gbits/s (6.6 Gbits/s was achieved for 10 flow matching on L2 header). While, OVS-DPDK shows better performance compared to the L2 tests, for instance for 64 bytes packets it reach 6.3 Gbits/s, almost catching up with VOSYSwitch value. Results for 100 flow entries are shown in Figure 9b, both switch solutions achieved the line rate with 256 bytes packets (128 bytes with VOSYSwitch for 2 cores). With high number of flow entries, a similar behavior is observed as for the L2 forwarding rate (Figure 8), i.e, OVS-DPDK keeps a significant rate while VOSYSwitch performance is going down because of the matching code growth. For low number of entries, VOSYSwitch has a better forwarding rate in both L2 and L3 cases, although, OVS-DPDK is more efficient with L3 flow entries than with L2.

V. Conclusion and Future Work

This paper proposes a novel OpenFlow software implementation, that uses an intermediate filtering language, namely the PCAP filter language, to dynamically classify network packet according to the entries defined in flow tables. Thanks to the just-in-time compiler LuaJIT and *pflua* library it provides a concise but feature full OpenFlow datapath bridge. Performance measurements have been performed, for simple flow configuration and low number of entries, VOSYSwitch is superior in particular in the case where the flow table match on L2 header. However, OVS-DPDK scales better, when a high number of flows are in use, it exhibits an almost steady rate. This is a research axis for the VOSYSwitch OpenFlow implementation on which performance improvements will be achieved, particularly by adding a per table caching mechanism. A cache will allow the Lua code generated by *pflua* to be only executed by the first packet of a flow, reducing the overhead introduced by a high number of flow entries.

Future works will include performance optimization of the OpenFlow datapath, but also an enhancement of the performance tests to cover use cases with more complex flow configurations, and to add more metrics such as the packet latency. Moreover, benchmarks related to the controller side, i.e., the OpenFlow channel, of the bridge will be performed.

Acknowledgment

This work has received funding from the European Union's Horizon 2020 research and innovation programme, 5GCity, under grant agreement No 761508.

References

[1] D. Kreutz, F. M. Ramos, P. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] "Network function virtualization (nfv) archtiectural framework," May 2015. [Online]. Available: http://www.etsi.org/deliver/etsi_gs/nfv/001_099/002/01.01.01_60/gs_nfv002v010101p.pdf

[3] N. B. Truong, G. M. Lee, and Y. Ghamri-Doudane, "Software defined networking-based vehicular adhoc network with fog computing," in *Integrated Network Management (IM), 2015 IFIP/IEEE International Symposium on*. IEEE, 2015, pp. 1202–1207.

[4] T. Subramanya, L. Goratti, S. N. Khan, E. Kafetzakis, I. Giannoulakis, and R. Riggio, "A practical architecture for mobile edge computing."

[5] Á. L. Valdivieso Caraguay, A. Benito Peral, L. I. Barona López, and L. J. García Villalba, "Sdn: Evolution and opportunities in the development iot applications," *International Journal of Distributed Sensor Networks*, vol. 10, no. 5, p. 735142, 2014.

[6] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 69–74, 2008.

[7] M. Paolino, J. Fanguède, N. Nikolaev, and D. Raho, "Turning an open source project into a carrier grade vswitch for nfv: Vosyswitch challenges & results," in *Network Infrastructure and Digital Content (IC-NIDC), 2016 IEEE International Conference on*. IEEE, 2016, pp. 22–27.

[8] "Snabb project code base," March 2018. [Online]. Available: https://github.com/snabbco/snabb

[9] M. Paolino, N. Nikolaev, J. Fanguède, and D. Raho, "Snabbswitch user space virtual switch benchmark and performance optimization for nfv," in *Network Function Virtualization and Software Defined Network (NFV-SDN), 2015 IEEE Conference on*. IEEE, 2015, pp. 86–92.

[10] "Openflow switch specification, version 1.3.5 (wire protocol 0x04)," 2015. [Online]. Available: https://www.opennetworking.org/software-defined-standards/specifications/

[11] "Pcap-filter man page," March 2018. [Online]. Available: https://www.tcpdump.org/manpages/pcap-filter.7.html

[12] "Packet filtering in lua," March 2018. [Online]. Available: https://github.com/Igalia/pflua
[13] "Luajit project home page," March 2018. [Online]. Available: http://luajit.org/
[14] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
[15] "Open vswitch," March 2018. [Online]. Available: http://openvswitch.org/
[16] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar *et al.*, "The design and implementation of open vswitch." in *NSDI*, 2015, pp. 117–130.
[17] "dpdk project website," March 2018. [Online]. Available: https://dpdk.org
[18] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle, "Performance characteristics of virtual switching," in *Cloud Networking (CloudNet), 2014 IEEE 3rd International Conference on*. IEEE, 2014, pp. 120–125.
[19] ——, "Assessing soft-and hardware bottlenecks in pc-based packet forwarding systems," *ICN 2015*, p. 90, 2015.
[20] R. Rahimi, M. Veeraraghavan, Y. Nakajima, H. Takahashi, S. Okamoto, and N. Yamanaka, "A high-performance openflow software switch," in *High Performance Switching and Routing (HPSR), 2016 IEEE 17th International Conference on*. IEEE, 2016, pp. 93–99.
[21] A. L. Aliyu, P. Bull, and A. Abdallah, "Performance implication and analysis of the openflow sdn protocol," in *Advanced Information Networking and Applications Workshops (WAINA), 2017 31st International Conference on*. IEEE, 2017, pp. 391–396.
[22] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, A. W. Moore *et al.*, "Oflops: An open framework for openflow switch evaluation." in *PAM*, vol. 7192. Springer, 2012, pp. 85–95.
[23] "Openstack networking neutron home page," March 2018. [Online]. Available: http://docs.openstack.org/developer/neutron/
[24] "Pflua benchmark," March 2018. [Online]. Available: https://github.com/Igalia/pflua-bench
[25] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "MoonGen: A Scriptable High-Speed Packet Generator," in *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, Oct. 2015.