Transforming a Traditional Home Gateway into a Hardware-accelerated SDN Switch

International Journal of Electrical and Computer Engineering (IJECE)
Vol. 10, No. 3, June 2020, pp. 2668~2681
ISSN: 2088-8708, DOI: 10.11591/ijece.v10i3.pp2668-2681

Transforming a traditional home gateway into a hardware-accelerated SDN switch

Sebastiano Miano, Fulvio Risso
Department of Computer and Control Engineering, Politecnico di Torino, Italy

Article Info

Article history:

Received May 15, 2019
Revised Oct 31, 2019
Accepted Dec 6, 2019

Keywords:

Hardware offloading
Openlow
Software defined networks

ABSTRACT

Nowadays, traditional home gateways must support increasingly complex applica-tions while keeping their cost reasonably low. Software Defined Networking (SDN) would simplify the management of those devices, but such an approach is typically reserved for new hardware devices, specifically engineered for this paradigm. As a consequence, typical SDN-based home gateway performs the switching in software, resulting in non-negligible performance degradation. In this paper, we provide our experience and findings of adding the OpenFlow support into a non-OpenFlow compatible home gateway, exploiting the possible hardware speedup available in the existing platform. We present our solution that transparently offloads a portion of the OpenFlow rule into the hardware, while keeping the remaining ones in software, being able to support the presence of multiple hardware tables with a different set of features. Moreover, we illustrate the design choices used to implement the func-tionalities required by the OpenFlow protocol (e.g., packet-in, packet-out messages) and finally, we evaluate the resulting architecture, showing the significant advantage in terms of performance that can be achieved by exploiting the underlying hardware, while maintaining an SDN-type ability to program and to instantiate desired network operations from a central controller.

Corresponding Author:

Sebastiano Miano,
Department of Control and Computer Engineering,
Politecnico di Torino,
Corso Duca degli Abruzzi, 24, 10129, TO, Italy.
Email: sebastiano.miano@polito.it

1. INTRODUCTION

Software Defined Networking (SDN) proposes a new paradigm into the computer networking field that allows network administrators to manage network services from a centralized point of control through abstraction of lower level functionality [1, 2]. This is achieved by separating the system that makes decisions about where traffic is sent (the control plane) from the underlying systems that forward traffic to the selected destination (the data plane). Of course, this approach brings significant benefits in the modern data center where the improved management of the network and the agility of re-configurations make this feature appealing and in great demand, given also the high number of servers that should be handled. At the same time, the SDN innovation can bring significant advancements to different areas too. An important domain that started to receive an increasing consideration is the administration of home networks [3].

Traditional home gateways are getting harder to manage as new application are introduced and moved at the customer premises. In this regard, applying SDN to this device would facilitate the home network management by providing the possibility to program and control the home network device from a centralize point of control, also allowing users to manage and configure the behavior of their network via high-level application that can be also designed by third-party developer [4]. To enable this paradigm shift, the most common way is to transform the home gateway into an OpenFlow [5] switch.

Even though over the years different hardware-based OpenFlow switches have been released [6] that perform very high-speed OpenFlow switching, the majority of Customer Premises Equipment's (CPEs) still use System on Chip (SoC) architectures with an integrated layer 2 device. Layer 2 switching is hardware-based, which means switches use application-specific integrated circuit (ASICs) for making switching decisions very quickly. They are usually traditional, non-OpenFlow compatible ASICs, which makes the transition to SDN- compliant solutions far away. Extending these devices with the support for the OpenFlow protocol would enable more flexible and granular service provisioning even to generally little organizations that couldn't manage the cost of upgrading their networks with new devices with native OpenFlow support.

A naive solution to this approach would be to add software SDN switch (e.g., Open vSwitch [7]) into the home gateway; in this case, the switching is done entirely in software, resulting in poor performance. Moreover, other high-speed packet processing solutions such as DPDK [8] or Netmap [9], would require to map one or more network interface to a CPU core, consuming important resources from the CPE that cannot be used to handle the other operations. To reduce this overhead and speedup the OpenFlow processing and forwarding of packet, we could exploit the hardware available in a traditional home gateway. Of course, not every operation can be accelerated; in fact, every hardware switch has a finite number of TCAM, critical for implementing line-speed forwarding, and it can hold only a limited number of flows. For some hardware, the number of flows is only one kind of limitation. Most switches were not designed with anything like OpenFlow in mind, especially when their interface ASICs were laid out. The chips do a excellent job of switching, and frequently handle basic Layer 3-4 functions as well, but OpenFlow asks for a great deal more.

This paper describes our experience in porting OpenFlow on already existing hardware switch with no support for the OpenFlow standard. We describe our architecture that integrates a hybrid software and hardware pipeline and can compensate the hardware limitations in terms of supported matches and actions, offloading only part of the OpenFlow rules, which are properly translated into the corresponding hardware related commands. While this mapping could result more vendor specific, we believe that the overall architecture for the offloading presented in this paper is vendor neutral enough to be exported in other platforms with similar characteristics. Moreover, we present our algorithm that is able to map the OpenFlow software pipeline into the corresponding hardware pipeline, considering the different number of tables (e.g., MAC, VLAN, ACL, etc..) available in the hardware device, while maintaining the semantic and precedence of the rules. In our work, we used the hardware pipeline of the NXP QorIQ T1040 platform, dividing the supported rules within the tables available in the integrated switch silicon. Finally, all the rules (including either the match or the action part) that cannot be mapped with the existing hardware, such as rewriting a MAC header, are executed in the software pipeline, which is based on the open source xDPd [10] project.

This paper is organized as follows: we describe the platform we used to validate our selective offloading algorithm in Section 2. Section 3 illustrates our architectural design for the OpenFlow rules offloading, and Section 4 presents the most significant implementation details of our prototype. Finally we show the evaluation and results in Section 5 and we briefly discuss related works in Section 6. Section 7 concludes the paper.

2. BACKGROUND

This section provides a description of the platform that we used to validate our solution. In particular, we present the interface and APIs that are used to program the behavior of the hardware switch available in the traditional home gateway together with a description of its hardware pipeline, whose knowledge is fundamental to correctly translate the set of OpenFlow rules into device-specific entries.

2.1. NXP QorIQ T1040

The NXP QorIQ T1040 [11] platform contains a four 64 bits CPU cores (PowerPC e5500), connected to additional modules belonging to the Data Path Acceleration Architecture (DPAA) [12] and peripheral network interfaces required for networking and telecommunications. It integrates a Gigabit Ethernet switch with eight 1 Gbps external ports and two internal 2.5 Gbps ports connected to Frame Manager (FMan) ports. In addition to typical Layer-2 operations, this module is also able to perform Layer 2-4 TCAM-based traffic classification on the ingress traffic, applying a specific set of actions (e.g., packet redirect) to the processed packet. The LAN ports of the L2 Switch make forwarding decisions based only on L2 switch logic, with no involvement from the FMan or CPU. As a consequence, the CPU cannot track the packets switched between the eight external L2 Switch ports, which might not be desirable in some use cases. To overcome this limitation, we adopt a specific mechanism to redirect these packets from the LAN ports to the CPU, which consists in using the copy_to_cpu flag as action in the ACL or MAC table;

this flag causes the copy of the packet matching this entry into an internal switch CPU queue. In this way, we can process the supported flows in hardware at line rate, while the unsupported ones will be redirected into the host's CPU where they can be processed by the software switch.

Access to L2 Switch PHY registers: A dedicated UIO Kernel Module, part of the NXP software development kit, maps L2 Switch and physical registers into user space, hence offering the possibility to program and control the behavior of the L2 Switch through sysfs entries. Notably, this kernel module avoids the commonly required context switching between kernel and userspace, because the device is accessed directly from user space.

L2 Switch APIs: The L2 Switch API represents a comprehensive, user-friendly and powerful function library that enables to program the switching module through high-level primitives, without managing singular registers. It incorporates the most common functions such as device initialization, port map setup, reset and configuration, including more complex functions used to specify the Quality of Service (QoS) configurations and Access Control Lists (ACLs) entries.

L2 Switch Hardware Pipeline: The NXP L2 Switch hardware pipeline is rather complex, as shown in the high- level view depicted in Figure 1. It is composed of a ingress and egress pipeline, both with a different set of tables and actions. When a packet arrives at a particular ingress port, after traversing the port MAC controller, it goes through the ingress pipeline, where is subjected to two classification steps. In the first (basic) classification stage, some basic information (e.g., VLAN tag, QoS class, DSCP value) are extracted from the packet and used in the next classification step. In the second, Advanced Multi-stage classification step, three TCAMs (named IS1, IS2 and ES0) serve different purposes. The IS1 table implements an L3-aware classification, allowing to override DSCP, QoS, VLAN ID values as a result of a lookup on L3-L4 headers. A second lookup is then made on the IS2 table, which applies typical ACL actions (i.e., permit, deny, police, redirect, mirror and copy to CPU) to the matched frame in a sequential order; if the packet matches the condition of a given ACL, the processing is stopped and the action is applied, otherwise it applies the default action, which usually drops all the traffic. Finally, in the egress pipeline, the ES0 table handles the egress forwarding based on VLAN and QoS policies. The size of these TCAMs is fixed, but the number of allowed entries depends on the complexity of each entry rule. As shown in Figure 1, the L2 forwarding module is based on a MAC Table supporting 8K entries; the L2 forwarding is done based on the VLAN classification, MAC addresses and the security enforcement as result of IS2.

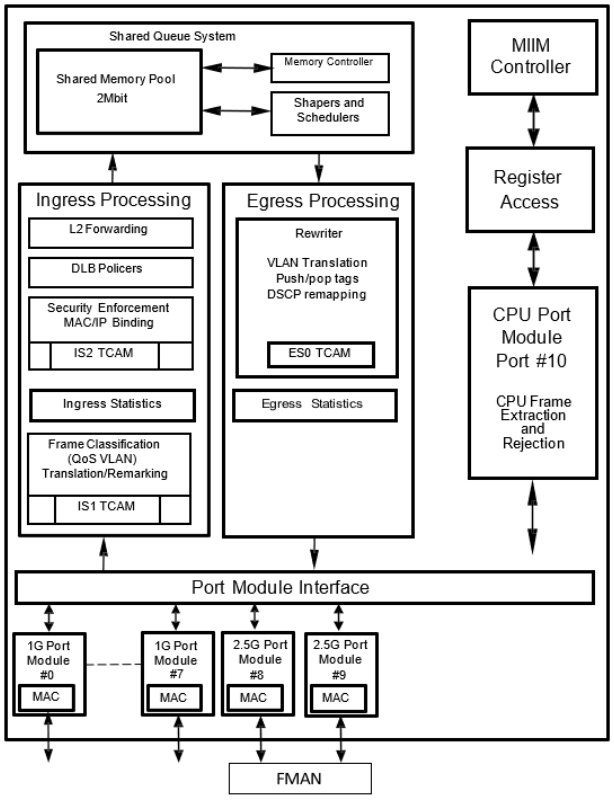

Figure 1. L2 switch hardware pipeline

2.2. xDPd software switch

The eXtensible DataPath daemon (xDPd) [10] is a multi-platform open-source datapath supporting multiple OpenFlow versions and built focusing on performance and extensibility, in particular with respect to (i) new hardware platforms (network processors, FPGAs, ASICs), (ii) new OpenFlow versions and extensions, and (iii) different management interfaces (e.g., OFConfig, CLI, AMQP, Netconf). The xDPd architecture, shown in Figure 2, includes a Hardware Abstraction Layer (HAL) that facilitates the porting of the OpenFlow pipeline on different hardware, hiding the hardware technology and vendor-specific features from the management and control plane logic. It uses the ROFL (Revised OpenFlow Library) libraries [13] as an HAL implementation and framework for creating OpenFlow agents communicating with different types of hardware platforms.

The ROFL library set is mainly composed of three different components. The *ROFL-common* library provides basic support for the OpenFlow protocol and maps the protocol's wire representation to a set of C++ classes. The *ROFL-hal* library provides a set of basic callback that should be implemented by the platform-specific driver to support the OpenFlow protocol features. Finally, the *ROFL-pipeline* library is a platform-agnostic OpenFlow 1.0, 1.2 and 1.3.X pipeline implementation that can be reused in several platforms. It is used as software OpenFlow packet processing library and serves as data-model and state manager for the ROFL-hal library.

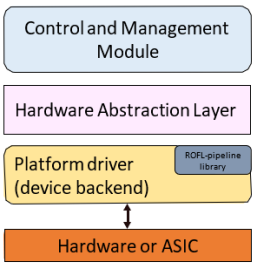

Figure 2. xDPd architecture

3. OVERALL ARCHITECTURE

Figure 3 describes the overall architecture of the system, which is made of a hardware *fast path* where packets are processed by the L2 switch integrated into the platform and a *slow path* that is running in software and is in charge of processing all packets whose matches or actions are not supported by the hardware. When a packet arrives at an ingress port, it is first processed by the hardware pipeline according to the rules installed by the *Selective Offloading Logic* component, if a matching rule is not found, packets are redirected to the NXP CPU where the xDPd software OpenFlow pipeline is running. It is worth mentioning that all the hardware details are hidden by an external user (e.g., OpenFlow controller), which programs the device as a simple OpenFlow switch; the rest of components will take care of translating the OpenFlow to match the hardware tables to speedup the packet processing.

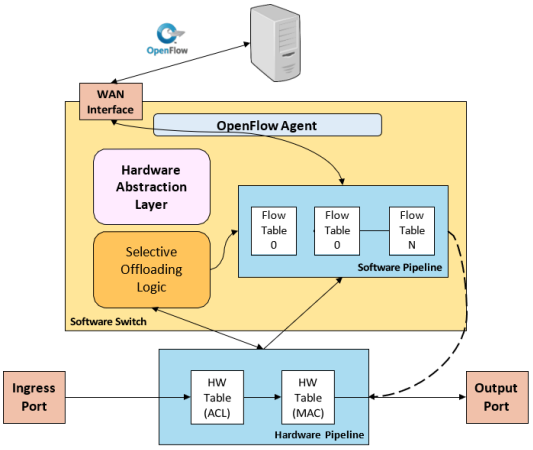

Figure 3. High-level design

3.1. The selective offloading logic component

The *Selective Offloading Logic* is the central component of the architecture and is in charge of managing the installation of the flow table entries, maintaining per port counters and translating the OpenFlow messages coming from the controller with the corresponding primitives required to interact with the hardware switch. This process typically involves deciding which flow entry the device can support (based on its feature set) and to sync the statistics from the device to the host. It consists of a northbound part that is responsible for the selection of the supported OpenFlow rules and a southbound side which is in charge of the communication with the device and is therefore strictly dependent on it.

The Northbound Interface should be aware of the switch pipeline capabilities, in particular regarding device tables, the match types and the actions allowed on each table. It maintains a data structure for each hardware table containing the device capabilities regarding supported matches and actions, which is used to check if a new flow rule is suitable for hardware offloading, e.g., if it's matching fields are a subset of the ones supported by the hardware. While the NB interface is generic enough for being exported from different devices (with similar hardware pipeline), the SB part should be changed to support the new device because it involves the flow entry insertion stage, which can obviously change depending on the underlying hardware switch. The description of how the communication with the hardware device has been implemented is described in Section 4.

3.2. Selection of the openflow rules for the offloading

An OpenFlow flow_mod message is used to install, delete and modify a flow table entry; as consequence, if the message contains a flow that is supported by the hardware device it should be offloaded accord- ingly. As show in the Algorithm 1, the first operation of the *Selective Offloading Logic* is to install the flow into the Software Switch table. This is done for two reasons. Firstly, the time required to insert a new rule into the software pipeline is usually faster than the one required to insert an entry into the hardware switch tables, since it is not affected by the other entries already installed into the forwarding tables. Secondly, this process would simplify the handling of *PacketOut* messages. Indeed, when a *PacketOut* is received, it should be injected into the data plane of the switch, carrying either a raw packet or indicating a local buffer on the switch containing a raw packet to release. Since the buffers are held by the software switch implementation and the software pipeline includes all the rules issued by the controller, its processing in the software pipeline is faster than injecting the packet in the hardware switch pipeline.

Algorithm 1 Selection of the rule to offload

```
1: procedure new_flow_mod (flow_entry_t* new_entry)
   add_entry_to_sw_table(new_entry);
2: if matches_supported(new_entry)
     && actions_supported(new_entry)
                    then
3:    offload(new_rule);
4: else
5:    if matches_supported(new_entry) &
      && actions_supported(new_entry) then
6:       new_entry.actions = copy_to_cpu;
7:       offload(new_entry);
8:    else
9:       if !matches_supported(new_entry) then
10:         for each rule in hwTables do
11:            if rules_set(rule) ⊆ rules_set(new_rule) then
12:               if check_correlation(new_rule, rule) then
13:                  delete_from_hardware(rule);
14:               end if
15:            end if
16:         end for
17:      end if
18:   end if
19: end if
20: end procedure
```

When the flow_mod is installed in the software table, it is checked to verify its suitability for the hardware offloading. In this case, the *Selective Offloading Logic* compares the matches and actions contained in the message with the data structure of each hardware table. If the flow includes matches and actions supported by the device tables the *Selective* Offloading *Logic* decides the right table (ACL or MAC table) in which place the rule (depending on their features set). Particularly, the new *flow_mod* is installed in the MAC-table if it contains only L2 dest MAC and VLAN as match criteria and the actions are the supported ones; redirect-to- port and send-to-controller. The remaining supported flows are placed in the ACL-table. After this process, the northbound interface calls the southbound part which takes care of install the specified rule in the hardware tables. However, if the hardware device supports the matches contained in the new entry but not its actions list, we need to process the packet matching that rule in the software pipeline. In this case, we inject the new rule in the hardware pipeline but with a single action to redirect the packet to the CPU, where the software pipeline processing applies the full action set contained in the original flow_mod. Finally, if the device does not support the new rule matches, to redirect the packets in the software pipeline, we should remove all hardware entries that interfere with the new rule matches set, avoiding that a new packet matches the hardware rule instead of the software one. When a correlated rule is discovered, it is deleted from the device tables so that a new packet will match the default rule that redirects all packets to the CPU for the software pipeline processing.

3.3. Mapping selected openflow rules on hardware tables

The Southbound Interface of the *Selective Offloading Logic* handles the mapping of the chosen OpenFlow rules in the hardware tables. This mapping is, of course, dependent on the underlying device. However, the organization of the MAC or ACL table is almost the same in all hardware switch ASICs, making the concepts applied to our offloading architecture also applicable to other architectures. If a flow can be offloaded in the MAC table, the corresponding hardware entry contains its fixed MAC address and VLAN ID. If the entry contains an output action to a specific port, the list of destination port in the hardware entry is filled with a Boolean value indicating if the packet should be forwarded to that particular port. The output_to_controller action is converted into an action with the copy_to_cpu flag enabled, indicating that the packet should be sent to a specific CPU queue and then redirected to the controller (how this task is achieved is specified in Section 4.). When a flow is offloaded to the ACL table, it is necessary to translate the formalism used by OpenFlow with the common fields contained in an ACL entry. The ACL uses a list of ports affected by that entry. In this case, if a rules specifies an ingress port, its corresponding Boolean value is enabled in that list. If not, the list includes all switch ports. An important consideration about this port list is required. Indeed, when an ACL rules include a behavior that also affects an output port, that port should also be added to the monitored port list. The actions supported by the ACL table are: permit, deny, redirect and copy to CPU. An OpenFlow drop action is translated in a *deny* action of the ACL, including a list of output ports for which the action should be applied. An OF output to port action is converted in a ACL *redirect* action, while the output to controller produces the enabling of the *copy_to_cpu* flag.

The process of moving a flow table entry to the hardware layer requires additional work if the table contains lower priority flow entries that (partially) overlap the newly installed flow entry. In these cases, together with the flow entry installation in the software layer, the *Selective Offloading Logic* decides to add them to the ACL table because the MAC table does not have a priority notion. Also, it performs an additional action that is the deletion of the flow table entries with lower priority, that are temporarily copied in the system's memory and the installation of the new flow entry with the other previously copied. On the other hand, if the new rule has a lower priority compared with those already installed in the ACL, it is inserted at the end of the list without moving the others. The flow table entry deletion from a hardware table is, in principle, a faster and simpler operation, while the installation requires a reorganization of the previously installed entries.

4. IMPLEMENTATION DETAILS

The xDPd/ROFL library set provides a Hardware Abstraction Layer that aims at simplifying the support of OpenFlow on a new platform. The *Platform Driver*, shown in Figure 2, includes the *Selective Offloading Logic* together with implementations for the buffer pool and the software pipeline used internally to simplify the OpenFlow porting of the NXP platform. The *Platform Driver*, also, uses the ROFL-pipeline library to implement an OpenFlow software switch and includes the logic to translate the OpenFlow messages coming from the controller in specific rules (if supported) for the hardware device. The main functionality provided by the driver, can be grouped in these 4 parts: (i) device and driver initialization, (ii) OpenFlow abstraction of the hardware switch, (iii) port status and statistics, (iv) packet-in and packet-out.

4.1. Device and driver initialization

The L2 Switch APIs provide an interface for accessing the physical registers of the underlying device, exposed to the user space applications through the kernel module described in Section 2.1. Writing these records allow us to program and control the behavior of the physical switch (insert flow rules, get statistics, change ports behavior, etc…). However, we also need to send/receive frames to and from each device port. The NXP Gigabit Ethernet switch core uses the MII (Media Independent Interface), which provides a Data interface to the Ethernet MAC for sending and receiving Ethernet frames, and a PHY management interface called MDIO (Management Data Input/Output) used to read and write the control and status registers. At start-up time, the driver performs some initialization steps. Firstly, it locates (usually under */dev/uioX*) and opens the UIO device, obtaining its file descriptor. Subsequently, it calls the *mmap* function to map the device memory into userspace, hence providing access to the device registers. In the end, the MDIO physical registers and devices are opened and used to read and write Ethernet frames from the physical ports.

4.2. OpenFlow abstraction of the hardware switch

An OpenFlow switch typically consists of several components. A virtual port module, which maps ingress and egress ports to some port abstraction, maintaining per-port counters; a flow table which performs lookups on flow keys extracted from packet headers; an action module, which executes a set of actions depending on the result of the flow table lookup. Our implementation mirrors these elements to allow the proposed selective offload. During the initialization phase, our device driver discovers the physical ports available in the hardware switch and adds them to the xDPd physical_switch structure, which represents a simple ab- straction used to control a generic switch while hiding platform-specific features. xDPd partitions the physical switch into Logical Switch Instances (LSIs), also known as *virtual switches*. In this driver we use a one-to-one mapping between the physical switch and a single LSI, hence mapping the physical ports directly to *Open- Flow physical ports*. Since the OpenFlow controller can add or remove an OpenFlow physical port from the LSI, the LSI may contain only a subset of the hardware switch ports.

4.3. Port management

The OpenFlow protocol includes also primitives to control and manage the status of the physical switch, such as reading the status of each port, add/modify/remove a port from the datapath, enable/disable forwarding, retrieve port statistics and more. The *Platform Driver* redirects these requests to the hardware switch once translated with the corresponding SDK API call. Furthermore, a controller can ask for port statistics (bytes received, dropped, etc…). Therefore the driver should read these statistics from the hardware switch and combine them with the similar stats of the software pipeline. As presented before, the OpenFlow physical ports of the LSI can be a subset of the hardware ports available in the switch; hence the *Platform Driver* keeps the explicit mapping between them, such as the fact that the hardware port #5 may actually corresponds to the OpenFlow port #2. When the controller sends a message referring to an LSI port, the driver retrieves the corresponding device port from an internal data structure and translates the OpenFlow command to the corresponding SDK API call. When the controller decides to modify the status of a single port, it sends an OpenFlow message that is received by the corresponding LSI. After parsing this message the ROFL-pipeline library call the corresponding method in the ROFL-hal, which should be implemented by the driver. When the driver receives this call, it can retrieve the corresponding port structure. However, this structure contains a port number that could be different from the physical port number. In order to retrieve the right physical port, ROFL-hal allows to add a platform specific structure to the switch port t. In this way, when we retrieve the port we have also its corresponding physical port number.

To provide a seamless compatibility with OpenFlow, the *Platform Driver* needs to implement also an asynchronous event handling mechanism, which is used to send the corresponding message to the OpenFlow controller (e.g., link detected, detached, etc. . .). However, while the SDK APIs provide several functions to query the switch for port status and statistics, they do not provide any asynchronous notification mechanism. Therefore, the *Platform Driver* uses a background task manager that checks every second the port status and, if necessary, notifies the xDPd Content and Management Module (CMM), which in turn passes this information to the OpenFlow controller. In short, the *background task manager* is used to check the following events: (i) expiration of a flow entry, (ii) free the space in the buffer pool when a packet becomes too old, (iii) update the port status and statistics and (iv) update the flow stats.

4.4. Packet-in and packet-out

Packet-In and Packet-Out messages are a fundamental feature of the OpenFlow protocol. The *Packet-In* enables a controller to receive packets from the OpenFlow switch as a result of a specific match-action tuple, which allows context-aware forwarding. Similarly, a *Packet-Out* message enables a controller to inject a particular packet into the switch, hence generating ad-hoc traffic for specific purposes (e.g., management).

Handling Packet-in messages: The generation of a *Packet-In* message is a consequence of a *redirect-to-controller* action in the flow table, which requires copying the packet from the physical switch to the *Platform Driver*. When a new flow mod containing the *redirect-to-controller* action is received, the *Selective Offloading Logic* converts that action into a hardware-dependent rule with the *redirect-to-cpu* flag enabled, which is supported by both ACL and MAC table. In this way, such a packet is no longer passing through the L2 switch; instead, it is delivered to the CPU port (the port #10 in Figure 1) and stored in a specific CPU queue, as shown in Figure 4. At this point, the *Platform Driver* can read the packet using the SDK APIs, hence triggering the generation of the appropriate OpenFlow message toward the controller. Packets that do not have to go to CPU ports are handled entirely by the switch logic and do not require any CPU cycles and happen at wire speed for any frame size. However, since the *Platform Driver* does not receive any notification when the packet reaches the CPU queue, a new background frame extractor thread has been created that polls continuously the CPU queues for new packets. When a new packet is detected, it generates a Packet-In message and sends it to the OpenFlow controller through the xDPd Control and Management Module. Packet-in messages can contain either the entire packet, or only a portion of it. In the latter case, the message will contain only the packet headers plus a BufferID (automatically generated by platform driver and opaque to the controller) that identifies the precise buffer that contains the actual (whole) packet. The controller can use the above BufferID when a packet-out is generated, telling that the packet under consideration is the one identified with the given BufferID. The driver locks any buffer currently in use, hence preventing it from being reused until it has been handled by the controller or a configurable amount of time has passed, avoiding zombies and memory exhaustion. Since the hardware switch does not have enough memory to store all the above packets, we move them in the memory buffer pool provided by xDPd, implemented in the device memory and linked to the corresponding LSI.

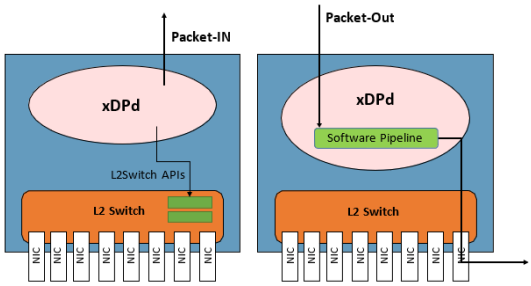

Figure 4. Packet-in and packet-out

Handling Packet-out messages: *Packet-Out* messages are used by the controller to force a specific packet (e.g., the one received via Packet-in) to be sent out of a specified port of the switch. These messages contain a full packet or a buffer ID referencing a packet stored in the buffer pool. The message must also include a list of actions to be applied in the order they are specified; an empty action list drops the packet. When the Packet- Out message contains an action list with only an output action, the packet is retrieved from the local buffer and injected, using the hardware switch APIs, into a physical port of the switch. Otherwise, the packet is injected directly into the software switch pipeline, which contains the whole set of flow rules, including the ones that are offloaded to the hardware. In this way the above packet will always cross only the software pipeline even if it is compatible with the rules present in the hardware pipeline; the limited generation rate of packet out messages makes this behavior insignificant from the performance perspective.

5. EVALUATION

In this section we evaluate the SDN capabilities of the traditional home gateway, which consist in receiving OpenFlow messages and configure the forwarding behavior of the data plane accordingly. In particular, in Section 5.1. we validate the proposed hybrid architecture where part of the rules are offloaded in the hardware pipeline (if supported) of the L2 switch available in the home gateway and the remaining rules are processed by the software OpenFlow switch. Moreover, in Section 5.2. we evaluate the capability of the traditional home gateway to accelerate the Ipv4 IPSec forwarding, which represent a valuable scenario in home and enterprise networks, where the traffic between two different OpenFlow switches should be encrypted to avoid the risk that physical devices in the path might read or write the contents of the tunnel packets.

5.1. xDPd driver evaluation

For this test we used the experimental setup depicted in in Figure 5. A workstation acting as both traffic generator (source) and receiver (sink) with the sufficient number of Gigabit Ethernet ports has been connected to the NXP hardware platform under test, i.e., the one presented in Section 2.1. Traffic is generated with the DPDK version of Pktgen [14], which has been modified in order to send traffic with the proper network parameters (e.g., MAC addresses) required by the specific test. In particular, we used DPDK 17.02.0-rc0 and Pktgen 3.0.17. In addition, a second workstation hosts the open source Ryu [15] OpenFlow controller, which is connected to a WAN port that is not terminated on the hardware switch of the NXP board. For our tests, we used the L2 learning switch application (called simpleswitch.py), whose behavior is the following: firstly, after the connect is established with the OpenFlow controller, the application installs a default rule in the flow table of the switch to redirect all packets to the controller. After that, the switch starts generating *Packet-In* messages for all packets received; then, the controller reads the source MAC addresses of the received packets and installs a new forwarding rule in the hardware switch as soon as a new MAC address is recognized (through a specific flowmod) message). The operating system of the T1040 is compiled with the version 1.7 of the NXP SDK and uses the Linux kernel version 3.12.19; xDPd has been installed in order to implement the OpenFlow pipeline. In fact, two different versions of xDPd are used; the first one is compiled with the support for our offloading driver, which can offload the supported rules to the hardware switch according to our proposed architecture. The second version is a vanilla xDPd compiled with the GNU/Linux driver, where the OpenFlow pipeline is implemented entirely in software.

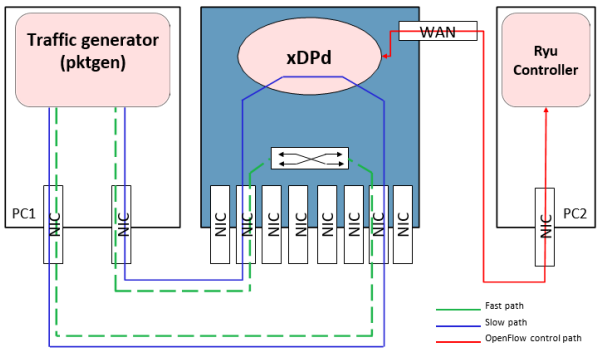

Figure 5. Test scenario with the ryu OF controller and the forwarding;
(a) with the implemented driver (green path), (b) with the software xDPd pipeline as a reference (blue path)

5.1.1. Experiment scenario

To understand the gain we can get from the hardware switching and processing, we performed 4 different experiments where we compare the application processing performance of a standard OpenFlow software switch implementation against our architecture that implements the proposed *Selective Offloading* algorithm. The goal of this experiment is not to show how the hardware can outperform the software, which is evident. Instead, we aim at demonstrating that (*i*) using the hardware available in the platform we can reduce the CPU processing that consequently becomes free for other tasks, and (*ii*) that we can introduce more flexibility to the platform, potentially enabling to support rules that are not natively supported in the hardware, while still leveraging the hardware for fast offloading, in a way that is completely transparent to the forwarding application. Our tests measure the throughput of the switch in two different operating conditions. First, we usedthe Port-based VLAN functionality, as described in Section 2.1. to redirect all the packets received by the hardware switch to the internal CPU, maintaining ingress port information and avoid switching in the L2Switch. This is used as a benchmarking, since it provides a simple way to introduce OpenFlow support in a traditional switch by moving all the processing in software. Second, we tested our offloading driver by selectively moving all the supported rules into the hardware switch, hence providing a more optimized way to bring OpenFlow support to an existing switching ASIC.

In the first two tests (Section 5.1.2. and 5.1.3.), we calculate the performance of the different driver implementation in terms of maximum throughput and we evaluate how this is affected by the number of ports involved into the processing and the size of the received packets. In particular, we report the throughput in million packets per seconds (Mpps) and the corresponding CPU and RAM consumption in the home gateway. On the other two tests (Section 5.1.4.), we take into account a scenario where only a part of the rules can be offloaded into the hardware device, while the unsupported ones are handled in the software OpenFlow pipeline.

5.1.2. SimpleSwitchapplication: forwarding between two ports

In the first test, PC1 and PC2 exchange a bidirectional traffic flow at the maximum speed (2 x 1Gbps). When the rules are installed correctly, the overall throughput of the system is depicted in Figure 6a, which shows that our driver leads to a significant performance improvement compared to the software-only version. In fact, we can notice that the line rate is never reached when the switching is performed entirely in software, likely due to the overhead caused by copying the packet data from user-space to kernel-space memory and vice versa. With our driver, the switching is performed entirely in hardware at line rate, as shown by the line associated to the throughput of the xDPd hardware, which is completely overlapped with the line rate.

5.1.3. SimpleSwitchapplication: forwarding between all ports

In the second experiment, we used a third machine PC3 equipped with a quad-port Intel I350 Gigabit Ethernet NIC, which was installed also in PC1. The four ports on PC1 are connected to the first four ports of the switch, while the remaining ports are attached to PC3. Both PC1 and PC3 generate bidirectional traffic using Pktgen DPDK at the maximum rate, with the same L2 Switch Ryu application used before. Results are shown in Figure 6b, with confirms that the hardware is still able to perform at line rate for whatever packet size, while the software is still very much beyond that throughput. It is worth noting that the line rate cannot be reached even in case of a more powerful CPU, as this component is connected to the switching hardware with a maximum aggregated bandwidth of 5Gbps, given by the two FMAN ports. Instead, the physical ports connected to the switch account for 8 Gbps of bidirectional traffic, i.e., 16Gbps, which is almost three time the capacity of the internal paths.

Figure 6c and 6d compares the CPU load between the xDPd GNU/Linux pure software implementation and the same values using the implemented driver. In the second experiment, where all ports receive frames at the maximum rate, the software xDPd implementation consumes almost all available CPU in the platform (4.0 on a quad core represents 100% utilization), given that every flow is handled by the system CPU. Comparing this result with the L2switch hardware driver confirms that, the use of the hardware device to perform the packet switching does not involve the CPU, which can be utilized by the other system processes. Of course, there are optimized OpenFlow switch implementations (OvS-DPDK or xDPd-DPDK) that use a software approach to obtain significant values of throughput. However, these technologies require too many resources (i.e., CPU cores) that would be prohibitive in a residential CPE, whose cost is a very important parameter to consider.

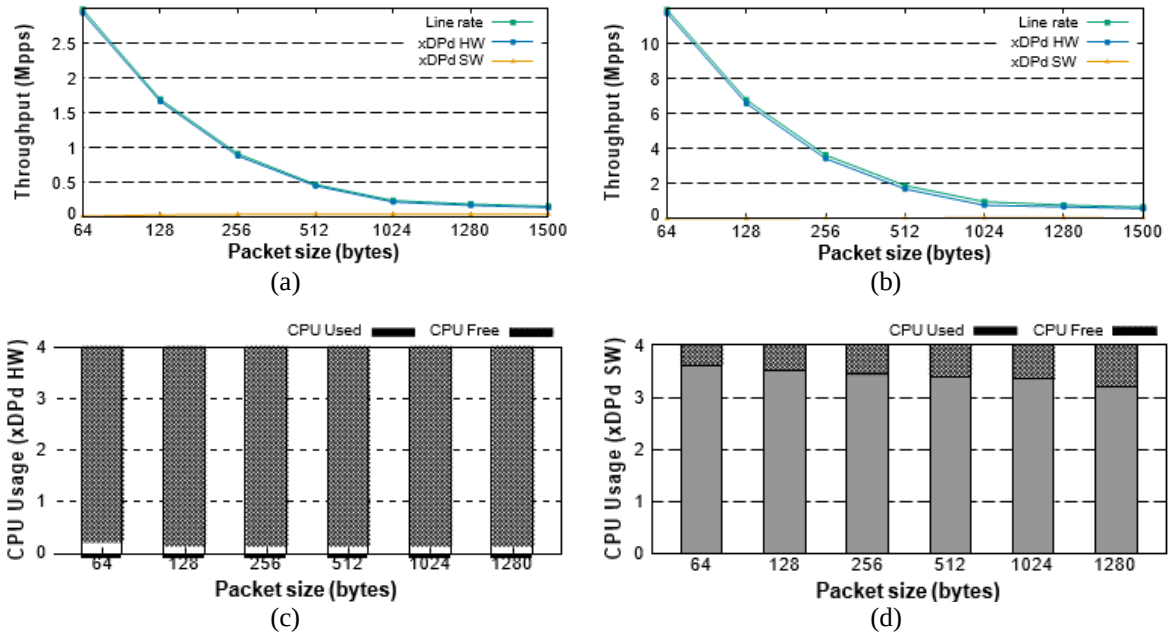

Figure 6. Performance comparison between xDPd with the hardware pipeline and software pipeline (SimpleSwitchOpenFlow application), (a) Forwarding between 2 ports, (b) Forwarding on all ports, (c) CPU consumption hardware pipeline, (d) CPU consumption software pipeline

5.1.4. FirewallRESTapplication: hybrid hardware-software processing

In this second set of tests we have deployed the Ryu firewallrest.pyapplication, which con- sists in a simple firewall that filters flows based on different type of matches on the packet headers (e.g., desti- nation IP address, Ethernet protocol type, TCP/UDP port). In particular, we evaluated two different scenarios where (i) the number of flow entries installed by the OpenFlow controller is equal to the maximum number of entries supported by out switch hardware tables (i.e., 2*K*) and (ii) the number of entries installed by the controller (i.e., 4*K*) exceeds the maximum number of supported flows in the hardware switch tables. In the latter case, only a specific set of rules is offloaded in the hardware switch, while the remaining ones are processed by the software switch pipeline. Moreover, it worth noting that in the fist scenario where the number of entries is equal to 2*K*, the number of offloaded rules can vary depending on the complexity of matches required by the OpenFlow controller since they will be offloaded in the hardware ACL table of the switch.

Figures 7 (a and b) show respectively the throughput achieved by our xDPd driver implementation (i.e., xDPd HW) against the software-only implementation (i.e., xDPd SW). Compared to the previous scenario, the forwarding throughput of this application is not equal to the maximum achievable throughput, since not all the rules can be offloaded into the hardware device; as consequence, the remaining rules will be handled by the software switch that is running in the device's CPU. This performance degradation is more evident when the number of installed rules increases; in this case, the slower processing in the software pipeline impacts more on the overall performance. Nevertheless, the advantages of exploiting the hardware pipeline are still conspicuous compared to the software pipeline both in terms of throughput and CPU consumption, as shown in Figure 7 (c and d).

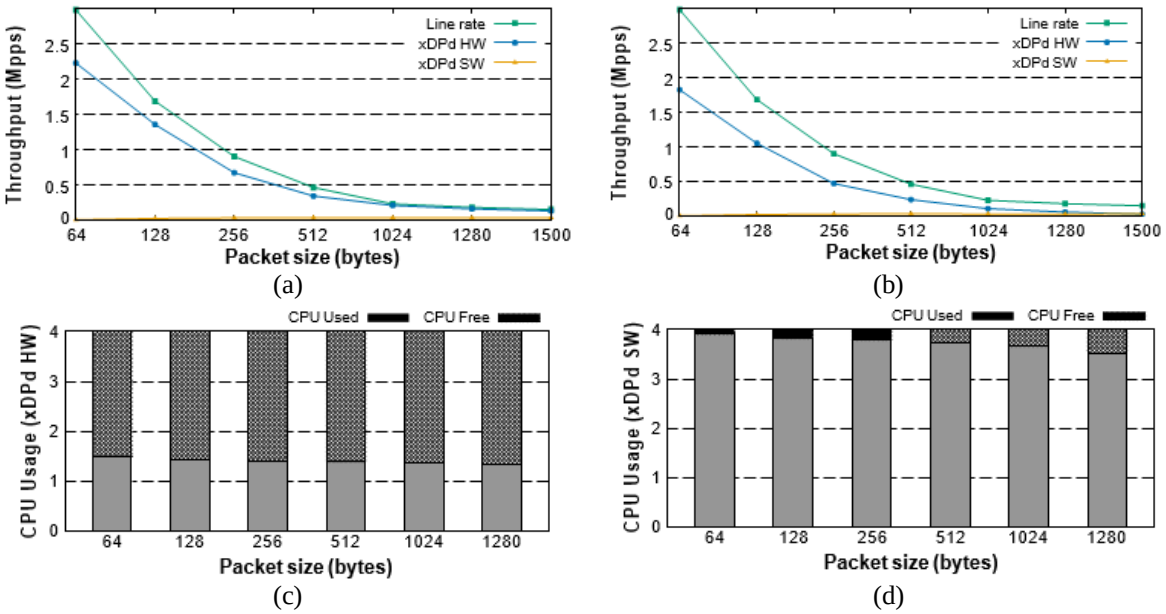

Figure 7. Performance comparison between xDPd with the hardware pipeline and software pipeline (FirewallRESTOpenFlow application), (a) Forwarding with 2K OpenFlow firewall rules, (b) Forwarding with 4K OpenFlow firewall rules, (c) CPU consumption hardware pipeline (4K rules), (d) CPU consumption software pipeline (4K rules)

5.2. IPSec hardware acceleration

In this test, we evaluate the capability of the device to handle IPSec traffic at high speed, given the hardware accelerations available in the device. In fact, a network administrator is usually connected to a CPE via an IPSec tunnel to safety the connection; IPsec encrypts IP payload and prevents the malicious party sniffing or manipulating the tunnel traffic. However, it introduces a non negligible overhead to the device that has to encrypt and decrypt the traffic, which is a costly operation. We demonstrate that this overhead can be significantly reduced by exploiting the hardware acceleration available within the device, in order to handle the IPSec tunnel in hardware without the intervention of the switch CPU that is free to handle other tasks.

5.2.1. Experiment scenario

The scenario used in this test is similar to the one depicted in Figure 5; the only difference is that two different home gateways (with our *Selective Offloading* architecture) were used. The first one receives the traffic from PC1, which acts as packet generator (using Pkgen-DPDK) and produces a stream of packets at different sizes, then it encrypts the traffic and forwards it through an IPSec ESP tunnel to the second T1040 device, which decrypts the traffic and forwards it back to the packet generator, which counts the total number of packets received. The Ryu controller is used to setup the tunnel between the two SDN-enabled CPEs.

5.2.2. Results

Figure 8 shows the maximum throughput achievable for the IPSec ESP tunnel scenario without the hardware acceleration (Figure 8a) and with the hardware acceleration (Figure 8b) available in the platform. As expected, the hardware acceleration provides a considerable improvement in the IPSec forwarding perfor- mance. In fact, we can notice that for packets larger than 256Bytes, the hardware acceleration can bring to line rate forwarding, while for smaller packets (i.e., 64 and 128Bytes) there is still a performance loss even with the acceleration enabled.

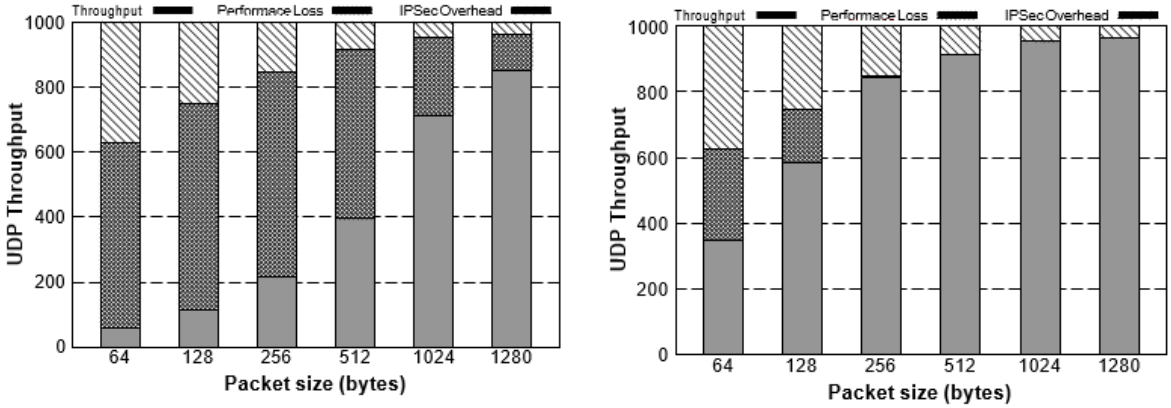

Figure 8. UDP Throughput comparison for IPSec tunnel encapsulation when using the hardware acceleration available in the device, (a) IPSec forwarding performance without acceleration, (b) IPSec forwarding performance with hardware acceleration

6. RELATED WORK

While OpenFlow is an evolving technology, a lot of attention has been paid to improve the OpenFlow switching performance using hardware components. Several works [16-21] focused on the idea of offloading OpenFlow packet processing from the host CPU level to onboard NIC hardware using FPGAs or Network Processors (NPs). Tanyingyong et al. [22] used a different approach based on a regular commodity Intel NIC rather than specialized NICs with FPGAs or NPs. In particular, they used the Intel Ethernet Flow Director component in the NIC [23], which provides filters that redirect packets, according to their flows, to queues for classification purposes, so as to be subsequently sent to a specific core into the host CPU for further processing. Although these works improved the lookup performance of the OpenFlow switching, they focused more on the software-based OpenFlow switching, as the only hardware-based feature used in the above prototypes was the hardware classifier available on selected network interface cards (NICs). During years, several companies tried to bring OpenFlow on their switch ASIC. The OpenFlow Data Plane Abstraction (OF-DPA) software defines and implements a hardware abstraction layer that maps the pipeline of Broadcom silicon switches to the OpenFlow 1.3.4 logical switch pipeline, utilizing the multiple device tables available in physical switches. This requires the controller to parse the hardware description contained in the Table Type Pattern (TTP) [24] to understand the capabilities and availability of hardware resources. Our work is based on a different concept. We expose to the controller a fully programmable OpenFlow switch, moving to our switch implementation the task of deciding which rules can be offloaded into the hardware. A similar idea has been presented by Netronome [25], which accelerates a software OpenFlow implementation (Open vSwitch) using a programmable network processor (NPU). However, being NPUs programmable, do not have the many limitations that we can encounter in existing

hardware switching ASICs. Finally, several commercial solutions are now available that transform selected hardware switch ASICs into OpenFlow-compatible devices. However, at the time of writing, no open source implementations are available that provide an insight about how this translation is done internally and how the OpenFlow messages (e.g. Packet-In, Packet-out,...) can be implemented in presence of an hardware pipeline.

7. CONCLUSION

In this paper we proposed an architecture to transparently offload OpenFlow rules into a traditional home gateway, allowing this device to receive SDN commands from a centralized controller and export the statistics of the matched traffic. This is done without upgrading the existing hardware and using a platform without SDN support. Our solutions is able to select the correct set of OpenFlow rules that can be offloaded into the hardware device, taking into account the different hardware pipeline of the L2-switch integrated into the platform. We perform a static analysis on the ruleset in order to keep the same priority and rule semantic of the original ruleset; if the result of the analysis is negative, the rule is not offloaded and continues to be handled into the software pipeline running on the device CPU.

We present the implementation details of an userspace driver for xDPd, a multi-platform OpenFlow switch, that accesses to the hardware switch registers to implement (when possible) forwarding decisions directly in the hardware pipeline, although the latter is not OpenFlow compliant. We illustrate the design choices to implement all the core functionalities required by the OpenFlow protocol (e.g., Packet-in, Packet-out mes- sages), and then we present an experimental evaluation of the performance gain we can achieve with the hard- ware switching and classification compared with the software-only counterpart. As expected, our driver imple- mentation shows a net performance advantage in terms of throughput and CPU consumption compared to the software-only implementation, thanks to its capability to exploit the existing non-OpenFlow hardware avail- able in the platform. Of course, the capability to exploit the hardware available depends also on the type of rules installed in the OpenFlow switch; in fact, if the OpenFlow ruleset contains a set of rules that cannot be offloaded into the device, our driver implementation redirects the conflicting packets into the software pipeline, where they will be processed by the slower CPU of the system. This can impact on the overall performance of the system, although it still retains better efficiency compared to the software-only approach. Moreover, we demonstrated that our architecture is also able to exploit other hardware speedup available in the traditional home gateway such as the IPSec encryption and decryption hardware module, which is transparently exploited when the OpenFlow controller sets up the appropriate tunnel. This performance gain is significant particularly in residential gateways where the limited resources can be a barrier for providing flexible network services, and that are so widely deployed in the nowadays Internet as home/business gateways that looks economically challenging to replace them with a new version with native OpenFlow support in hardware.

REFERENCES

[1] Yap KK, Huang TY, Dodson B, Lam MS, McKeown N., "Towards software-friendly networks," *In Proceed-ings of the first ACM asia-pacific workshop on Workshop on systems*, pp. 49-54, 2010.
[2] Kreutz, D., Ramos, F., Verissimo, P., Rothenberg, C.E., Azodolmolky, S. and Uhlig, S., "Software-defined networking: A comprehensive survey," *arXiv preprint arXiv: 1406.0440*, 2014.
[3] Alshnta, A. M., Abdollah, M. F., and Al-Haiqi, A., "SDN in the home: A survey of home network solutions using Software Defined Networking," *Cogent Engineering*, vol. 5, no. 1, 1469949, 2018.
[4] Bonafiglia R, Miano S, Nuccio S, Risso F, Sapio A., "Enabling NFV services on resource-constrained CPEs," *In 2016 5th IEEE International Conference on Cloud Networking (Cloudnet)*, pp. 83-88, 2016.
[5] McKeown N, Anderson T, Balakrishnan H, Parulkar G, Peterson L, Rexford J, Shenker S, Turner J., "OpenFlow: enabling innovation in campus networks," *In ACM SIGCOMM Computer Communication Review*, pp. 69-74, 2008.
[6] Hardware OpenFlow Switches, [Online]. Available: *https://www.opennetworking.org/sdn-openflow-products*
[7] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J. and Amidon, K., "The design and implementation of open vswitch," In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pp. 117-130, 2015.
[8] Data Plane Development Kit. [Online]. Available: *https://www.dpdk.org/*
[9] Rizzo, L., "Netmap: a novel framework for fast packet I/O," In *21st USENIX Security Symposium (USENIX Security 12)*, pp. 101-112, 2012.
[10] Suñé, M and Köpsel, A and Alvarez, V and Jungel, T., "xDPd: eXtensible DataPath Daemon," In *EWSDN, Berlin, Germany*, 2013.
[11] NXP, "QorIQ T1040 and T1020 Processors," NXP, 2015. [Online]. Available: *https://bit.ly/2nPRsBp*
[12] NXP, "QorIQ Data Path Acceleration Architecture," NXP, 2014. [Online]. Available: *http://cache.freescale.com/files/training/doc/ftf/2014/FTF-NET-F0146.pdf*

[13] ROFL, Revised OpenFlow Library, Berlin Institute for Software Defined Networks (BISDN), [Online]. Available: *https://web.archive.org/web/20191219112812/https://bisdn.github.io/rofl-core/rofl-common/index.html*
[14] "*Packet Gen*-erator with DPDK," pktgen, [Online]. Available: *https://pktgen-dpdk.readthedocs.io/en/latest/*
[15] OSRG, "Ryu SDN Framework," OSRG, [Online]. Available: *https://osrg.github.io/ryu/*
[16] Naous, J., Erickson, D., Covington, G. A., Appenzeller, G., McKeown, N., "Implement- ing an OpenFlow switch on the NetFPGA platform," In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 1-9, 2008.
[17] Luo, Y., Cascon, P., Murray, E., and Ortega, J., "Accelerating OpenFlow switching with net- work processors," In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pp. 70-71, 2009.
[18] Tanyingyong V, Hidell M., and Sjödin P., " Improving pc-based openflow switching performance," *In 2010 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 1-2, 2010.
[19] Suñé, M., Alvarez V. Jungel T., Toseef U., Pentikousis K., "An OpenFlow implementation for network processors," *In EWSDN*, pp. 123-124, 2014.
[20] SDN Enabled CPE (Smart Traffic Steering), [Online]. Available: *https://noviflow.com/smart-traffic-steering/*
[21] Pan, H., Guan, H., Liu, J., Ding, W., Lin, C., and Xie, G., "The FlowAdapter: Enable flexible multi-table processing on legacy hardware," *In Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pp. 85-90, 2013.
[22] Tanyingyong, V., Hidell, M., and Sjödin, P., "Using hardware classification to improve pc-based openflow switching. In *High Performance Switching and Routing (HPSR), 2011 IEEE 12th International Conference*, pp. 215-221, 2011.
[23] Intel Corporation, "Intel ethernet flow director and memcached performance White Paper," Intel Corporation, 2014. [Online]. Available: *https://intel.ly/2nQ4m2i*
[24] Nabil Damouny, *et al.*, "Simplifying OpenFlow Interoperability with Table Type Patterns (TTP)," *Open Networking Foundation (ONF)*, 2015.
[25] Rolf Neugebauer, "Netronome. Selective and transparent acceleration of OpenFlow switches," *Netronome*, 2013.

BIOGRAPHIES OF AUTHORS

Sebastiano Miano is pursuing his Ph.D. degree at Politecnico di Torino, Italy, where he received his Master's degree in Computer Engineering in 2015. His research interests include programmable data planes, software defined networking and high-speed network function virtualizations.

Fulvio Risso received the M.Sc. (1995) and Ph.D. (2000) in computer engineering from Politecnico di Torino, Italy. He is currently Associate Professor at the same University. His research interests focus on high-speed and flexible network processing, edge/fog computing, software-defined net- works, network functions virtualization. He has co-authored more than 100 scientific papers