Partial Offloading of OpenFlow Rules on a Traditional Hardware Switch ASIC

Sebastiano Miano*, Fulvio Risso*
*Politecnico di Torino, Dept. of Computer and Control Engineering
Corso Duca degli Abruzzi 24, 10129 Torino, Italy

Hagen Woesner†
†Berlin Institute for Software Defined Networks
Christburger Straße 45, 10405 Berlin, Germany

***Abstract*—OpenFlow represents a new powerful paradigm that combines the flexibility of the software with the efficiency of a programmable hardware switch. However, such an approach is currently reserved for new hardware devices, specifically engineered for this paradigm.**

This paper presents our experience and findings about selectively offloading OpenFlow rules into a non-OpenFlow compatible hardware switch silicon, which enables existing (legacy) hardware ASICs to become compatible with the SDN paradigm. We describe our solution that transparently offloads the portion of OpenFlow rules supported by the hardware while keeping in software the remaining ones, and that is able to support the presence of multiple hardware tables although with limited capabilities in terms of matches and actions. Moreover, we illustrate the design choices used to implement all the basic functionalities required by the OpenFlow protocol (e.g., packet-in, packet-out messages) and then we demonstrate the considerable advantage in terms of performance that can be obtained by performing switching in hardware, while maintaining an SDN-type ability to program and to instantiate desired network operations from a central controller.

***Index Terms*—Software Defined Networks, OpenFlow, Hardware Offloading.**

I. Introduction

Although over the years several hardware-based OpenFlow switches have been released [1] that perform very high-speed OpenFlow switching, the majority of Customer Premises Equipments (CPEs) such as residential gateways use System on Chip (SoC) architectures with an integrated layer 2 device, and are ideally suited for use in mixed control and data plane applications. 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.

Bringing OpenFlow on these devices, without the need of changing the existing hardware, enables more flexible and granular service provisioning even to relatively small companies that could not afford the effort required to upgrade their network with new devices that support natively the OpenFlow protocol.

In their natural mode of operation, traditional switching ASICs can move packets between all of their ports at full line rate, so it is a reasonable assumption that they can do the same when used with OpenFlow. Sadly, that is not the case. 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. Moreover, 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 presents our experience in porting OpenFlow on already existing hardware switch with no support for the OpenFlow standard. We describe our solution for compensating the hardware limitations in terms of supported matches and actions, offloading only part of the OpenFlow rules that can be handled by the hardware. Of course, this offloading requires a translation between the OpenFlow specific messages to hardware related commands, which means to take the OpenFlow commands and map them to API calls inside of the switch. 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, hardware switches have multiple tables (e.g., MAC, VLAN, ACL, etc. . .), hence offloading OpenFlow rules on the switch require to fit them into the existing hardware pipeline, taking into account the precedence between rules in order to preserve their semantic. 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 project.

This paper is organized as follows: we briefly discuss related works in Section II, while Section III describes the architecture of the platform used to validate our selective offloading algorithm. Section IV illustrates our architectural design for the OpenFlow rules offloading, and Section V presents the most significant implementation details of our prototype. Finally we show the evaluation and results in Section VI and conclude the paper in Section VII.

978-1-5090-6008-5/17/$31.00

II. 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 [2, 3] 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.* [4] 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, 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) [5] 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 [6], 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.

III. Background

This section provides a description of the architecture we based our solution on, including the way we can program its PHY registers and the structure of its hardware pipeline, which is required to process the mapping between OpenFlow rules and device-specific entries.

A. NXP QorIQ T1040

The NXP QorIQ T1040 platform contains a four 64 bits CPU cores (PowerPC e5500), connected to additional modules belonging to the Data Path Acceleration Architecture (DPAA) [7] and peripheral network interfaces required for networking and telecommunications.

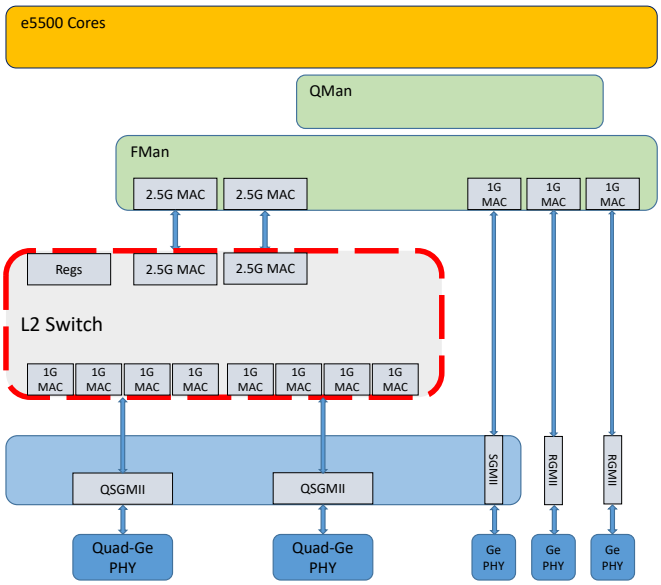

Fig. 1: NXP QorIQ T1040 platform

The above platform integrates also a Gigabit Ethernet switch core that supports eight 1 Gbps external PHY ports and two internal 2.5 Gbps ports connected to Frame Manager (FMan)[1] ports, as shown in Figure 1. While the switching core operates at layer two, this module integrates also a Layer 2-4 TCAM-based traffic classifier operating on the ingress traffic, which can select and perform basic actions on the incoming traffic (e.g., packet redirect) based on information such as MAC addresses, EtherType, VLAN tags, IP addresses, DSCP and TCP/UDP ports and ranges, as detailed in Figure 2. The L2 switch core supports wire-speed, hardware-based learning, and CPU-based software learning that is configurable per port.

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 can use the Port-based VLAN feature, which assigns a VLAN to all packets coming from a given physical port, hence avoiding switching in the L2 Switch and preserving ingress port information. However, this introduces two additional issues: (*i*) if a packet arrives at an external port already tagged, it will be classified based on the VLAN ID specified in the VLAN tag, ignoring the port-based VLAN setting; and (*ii*) the aggregated throughput of all eight external ports cannot be higher than 5 Gbps due to the speed of the internal ports. Moreover, given that all frames received by the LAN ports will be redirected to the CPU and processed

[1]The Frame Manager is a component of the DPAA architecture which combines the Ethernet network interfaces with packet distribution logic to provide intelligent distribution and queuing decisions for incoming traffic.

by the software switch logic, we may not be able to manage the received frames at the same speed of the hardware switch.

*1) **Access to L2 Switch PHY registers**:* A dedicated UIO Kernel Module[2], part of the NXP software development kit, maps L2 Switch and PHY 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. This enables programmers to define optimized access to the hardware switch traffic, sending/receiving frames through a *char* device.

*2) **L2 Switch APIs**:* The L2 Switch API represents a comprehensive, user-friendly and robust function library that enables to program the switching module through high-level primitives, without having to deal with individual registers. It includes the most common functions such as device initialization, port map setup, port reset and configuration, port status polling and configuration based on auto-negotiation, mirroring, link aggregation, port VLAN statistics, Quality of Service (QoS) configurations and Access Control Lists (ACLs).

*3) **L2 Switch Hardware Pipeline**:* The NXP L2 Switch hardware pipeline is rather complex, as shown in the high-level view depicted in Figure 2. When a packet arrives at a particular ingress port, the MAC controller of the port checks the Frame Check Sequence (FCS) and the VLAN tag size, if they are not valid, the frame is discarded without further processing.

After traversing the port MAC controller, the packet goes through the *Ingress Processing* pipeline, where is subjected to two classification steps. In the first (basic) classification stage, the packet is accepted if contains a valid VLAN tag and valid MAC addresses. During this stage, some basic information (VLAN tag, QoS class, DP level, 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. In particular it is possible to add rules that modify QoS parameters (e.g. DSCP, PCP, DP, ...) or set the VLAN ID for a matching frame. 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. Packets are matched against ACL rules in a strictly sequential order, which means that when a packet matches the condition of a given ACL, the processing is stopped and the action contained in the matched rule is enforced. If no matches are found a default action is applied, which usually denies the traffic. Finally, 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.

[2]The Userspace I/O framework (UIO) allows developers to write user-space-based device drivers. It defines a small kernel-space component that indicates device memory regions to user space and provides interrupt indication to user space.

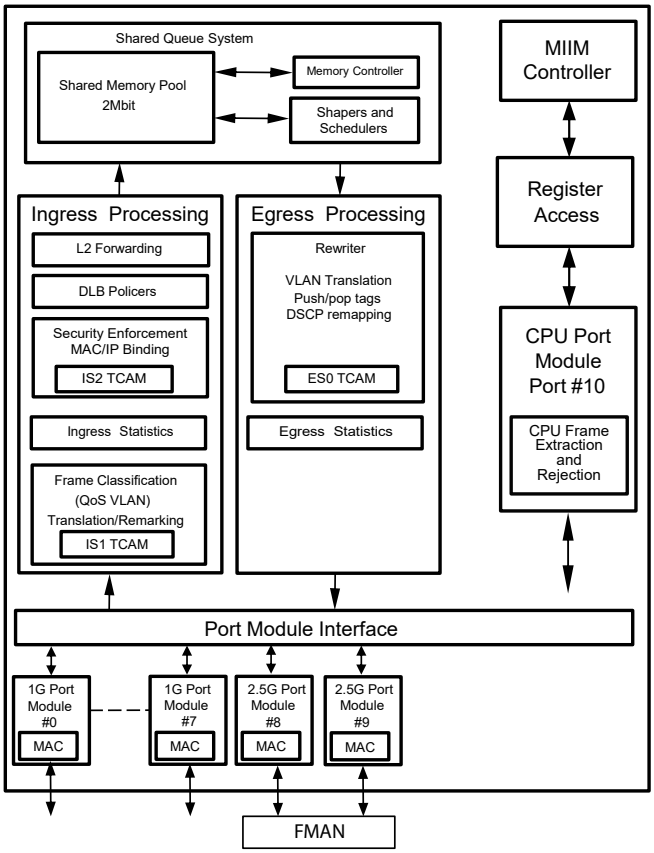

Fig. 2: L2 switch hardware pipeline

As shown in Figure 2, 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*.

B. xDPd Software Switch

The eXtensible DataPath daemon (xDPd) [8] 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 (OFConfig, CLI, AMQP, Netconf...).

The xDPd architecture, shown in Figure 3, includes a Hardware Abstraction Layer (HAL) that facilitates the porting of the OpenFlow pipeline on different hardware. In fact, this hides the hardware technology and vendor-specific features from the management and control plane logic, hence allowing to return a simple OpenFlow switch to upper layers. It uses the ROFL (Revised OpenFlow Library) libraries [9] 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 sup-

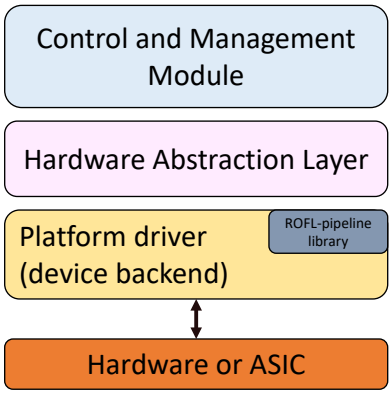

Fig. 3: xDPd architecture

port for the OpenFlow protocol and maps the protocols 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.

IV. OpenFlow Rules Offloading Architecture

This section presents the architecture that has been defined to offload OpenFlow rules on our traditional (non-OpenFlow) switching ASIC, which is shown in Figure 4. Briefly, a high-performance hardware switch is introduced in the switch's fast path, that is, the part of the switch that performs packet forwarding operations, while a software switch is in charge of processing all the packets whose matches or actions are not supported by the hardware. When a packet arrives at an ingress port, it goes through the hardware pipeline (Section III-A3) and is processed according to the installed rules. The hardware is programmed in such a way that packets that cannot be handled by the L2 switch will match a default entry in the ACL table, that will redirect the packet to the NXP CPU where is processed by the xDPd software switch.

A. The Selective Offloading Logic component

The *Selective Offloading Logic* represents the central component of the architecture and is located in the slow path of the system. It manages the installation of the flow table entries, maintains per port counters and translates 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 its 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 V.

B. Selection of the OpenFlow rules for the offloading

An OpenFlow *flow_mod* message is used to install, delete and modify a flow table entry. The Algorithm 1 shows the pseudo-code used to select the rule suitable for the hardware offloading. When a new *flow_mod* comes from the OpenFlow controller, it is always installed in the Software Switch table. The reason for this is twofold. Firstly, the installation time in the software table is usually faster than the hardware because it is not affected by the other entries already installed in the forwarding tables. Secondly, having all rules in the software pipeline help us to process the received *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.

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,

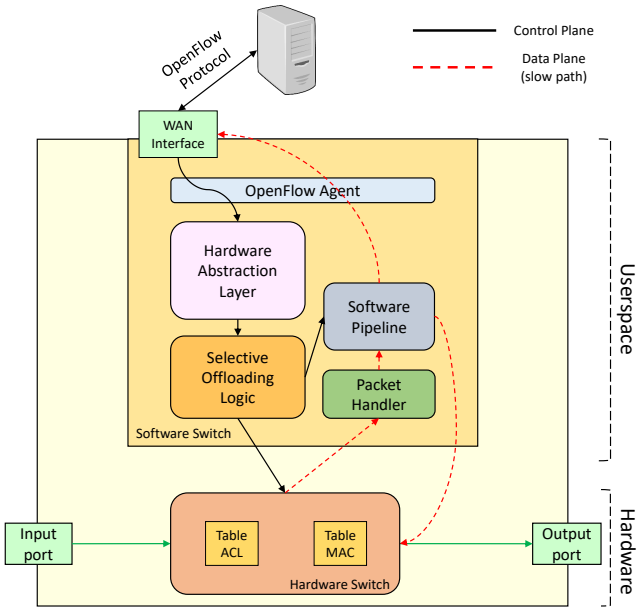

Fig. 4: High-level design

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.

Algorithm 1 Selection of the rule to offload

```
1:  procedure flow_mod_add (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
```

C. 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 V).

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.

V. 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 3, 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.

A. 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 III-A1. 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.

B. 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 abstraction 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[3], hence mapping the physical ports directly to *OpenFlow physical ports*[4]. 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.

C. 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.

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.

D. 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).

[3]The current prototype does not support multiple LSI levels. This extension is left for future work.

[4]The OpenFlow physical ports correspond to a hardware interface of the switch. For example, on an Ethernet switch, physical ports map one-to-one to the Ethernet interfaces. The current prototype does not support OpenFlow *virtual* ports.

1) 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 2) and stored in a specific CPU queue[5], as shown in Figure 5. 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.

2) 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]The NXP platform defines eight different hardware memory pools that are associated to the internal CPU; when the packet is transfered to the CPU port, we have to specify which one among the above queues has to be used.

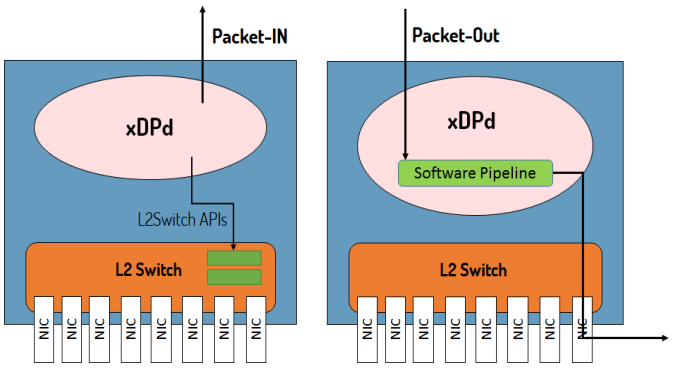

Fig. 5: Packet-in and Packet-out

VI. Evaluation

A. Experiment setup

The proposed architecture has been evaluated with the experimental setup depicted in Figure 6. 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 III-A. Traffic is generated with the DPDK version of Pktgen [10], 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 [11] 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, which reads the source MAC addresses of the received packets (sent to the OpenFlow controller using Packet-in messages) and installs a new forwarding rule in the hardware switch as soon as a new MAC address is recognized.

B. Experiment scenario

To investigate how much we gain from offloading the lookup and switching process task to the hardware switch, we carried out two experiments to compare the application processing performance of a standard software switch implementation (xDPd with the GNU/Linux driver) and our driver that implements the proposed *Selective Offloading* architecture. 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 flexibiliby to the platform, potentially enabling to support rules that are not natively supported in the hardware, while

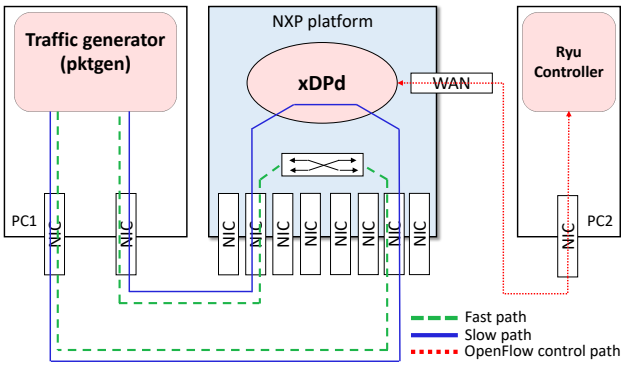

Fig. 6: Test scenario with the Ryu OF controller and the forwarding: (a) with the implemented driver (green path), (b) with the software xDPd pipeline as a reference (blue path).

TABLE I: CPU load and RAM utilization with the xDPd GNU/Linux driver and the L2Switch hardware driver.

Pkt size	Software pipeline		Hardware pipeline	
(bytes)	CPU load	RAM (MB)	CPU load	RAM (MB)
64	3.61 / 4	540	0.21 / 4	360
128	3.52 / 4	532	0.13 / 4	350
256	3.44 / 4	532	0.12 / 4	341
512	3.39 / 4	532	0.12 / 4	340
1024	3.36 / 4	532	0.12 / 4	334
1280	3.20 / 4	532	0.12 / 4	320
1500	3.19 / 4	532	0.12 / 4	320

still leveragin 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 used the Port-based VLAN functionality, as described in Section III-A, which redirects 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.

We performed two different tests to assess the performance of the xDPd GNU/Linux driver and compare them with the implemented offloading driver. We are interested in both the maximum throughput and in how it is affected by the number of ports involved and the packet size, with the companion CPU consumption. The throughput is measured in million packets per second (Mpps), for various packet sizes.

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 7, which shows that our driver leads

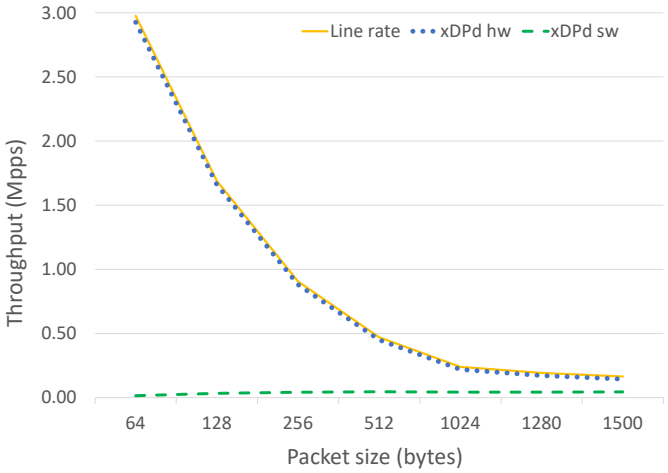

Fig. 7: Forwarding between 2 ports

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.

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 8, 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 shown in Figure 1. 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.

Table I compares the CPU load and RAM consumption 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

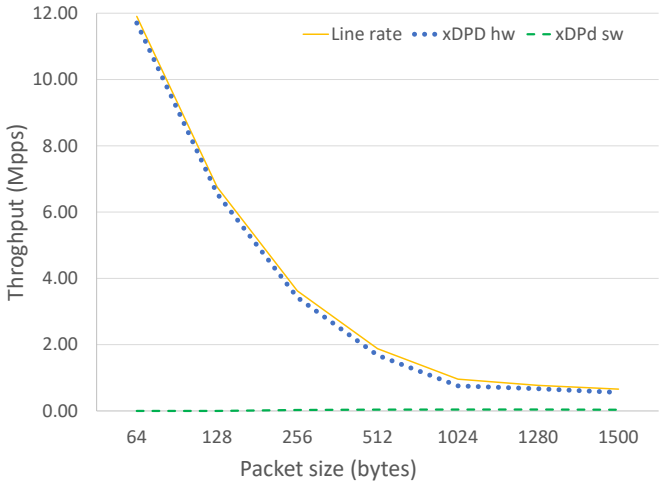

Fig. 8: Forwarding on all ports

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.

VII. Conclusion and Future Work

This paper proposes an architecture for the *Selective Offloading* of OpenFlow rules into non-OpenFlow compatible hardware ASICs, to enable existing (legacy) network equipment to support the OpenFlow protocol.

Our solution is based on selecting the right OpenFlow rules suitable for the hardware offloading while maintaining the unsupported ones in the software pipeline. Specifically, we offload the chosen rules into the hardware switch pipeline and in particular into the ACL and MAC table, performing the conversion between OpenFlow rules to the table-specific matches and actions.

We outline a design architecture and 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 messages), and then we present an experimental evaluation of the performance gain we can achieve with the hardware switching and classification compared with the software-only counterpart.

As expected, the implemented driver shows a considerable advantage with respect to throughput and CPU consumption, thanks to its capability to exploit the existing non-OpenFlow hardware available in the platform. 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.

The idea of splitting the rules across the hardware and software pipelines is particularly suitable for OpenFlow 1.0, which operates with a single logical table. In this case, flow rules are inspected and offloaded to the hardware according to the capabilities of the ACL and MAC tables. In the case of a logical pipeline that includes multiple tables, only the flow rules that are installed in the first table are potentially offloaded to the hardware switch.

A future version of the *Selective Offloading Logic* could be to incorporate the work of [13] in the current architecture, which enables the convertion of a M-stage pipeline (with M flow tables) into a N-stage pipeline, hence adapting the logical pipeline to the physical pipeline of the hardware. In this case, the *Flow Adapter* could handle the conversion between the software and the hardware pipelines and the *Selective Offloading Logic* could maintain the ability to select which rules can be offloaded and how to translate them to fit into the actual hardware device.

References

[1] Hardware OpenFlow Switches, [Online] https://www.opennetworking.org/sdn-openflow-products?start=20

[2] Naous, J., Erickson, D., Covington, G. A., Appenzeller, G., McKeown, N. (2008, November). Implementing 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). ACM.

[3] Luo, Y., Cascon, P., Murray, E., Ortega, J. (2009, October). Accelerating OpenFlow switching with network processors. In *Proceedings of the 5th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (pp. 70-71). ACM.

[4] Tanyingyong, V., Hidell, M., Sjdin, P. (2011, July). Using hardware classification to improve pc-based openflow switching. In *High Performance Switching and Routing (HPSR), 2011 IEEE 12th International Conference* on (pp. 215-221). IEEE.

[5] Table Type Pattern, [Online] https://web.archive.org/web/20161110034114/https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/OpenFlow\%20Table\%20Type\%20Patterns\%20v1.0.pdf

[6] Rolf Neugebauer - Netronome. Selective and transparent acceleration of OpenFlow switches

[7] QorIQ Data Path Acceleration Architecture, [Online] http://www.nxp.com/products/microcontrollers-and-processors/power-architecture-processors/qoriq-platforms/data-path-acceleration:QORIQ_DPAA

[8] Suñé, M and Köpsel, A and Alvarez, V and Jungel, T. xDPd: eXtensible DataPath Daemon. In *EWSDN, Berlin, Germany, 2013*

[9] ROFL, [Online] https://www.codebasin.net/redmine/projects/rofl-core/wiki/Wiki?version=12

[10] *Packet Gen*-erator with DPDK, [Online] https://pktgen.readthedocs.io/en/latest/index.html

[11] Ryu SDN Framework, [Online] https://osrg.github.io/ryu/

[12] SDN Enabled CPE (Smart Traffic Steering), [Online] http://noviflow.com/solutions/sdn-enabled-cpe-smart-traffic-steering/

[13] Pan, H., Guan, H., Liu, J., Ding, W., Lin, C., & Xie, G. (2013, August). 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). ACM.