Programmable Abstraction of Datapath

Advanced Programmability of Heterogeneous Datapath Elements through Hardware Abstraction

Bartosz Belter, Artur Binczewski, Krzysztof Dombek, Artur Juszczyk, Łukasz Ogrodowczyk, Damian Parniewicz, Maciej Stroiński
Poznan Supercomputing and Networking Center
Poland
{bartosz.belter, artur, kdombek, juszczyk, lukaszog, damianp, stroins} at man.poznan.pl

Iwo Olszewski
Independent researcher
Poland
ict at iwolszewski.pl

***Abstract*— Despite continuous developments in this area, Software Defined Networking (SDN) still seeks for a flexible way of defining a network device behaviour. The control plane needs to be able to fully utilize growing capabilities of modern networking hardware and its diversity. In this paper we propose a new hardware abstraction for various network devices (network processors, optical devices and access devices). The first goal of this proposal is to expose advanced programmability capabilities of network processors and software switches. The second goal of our proposal is to extend the concept of the network node programmability by giving a possibility to dynamically check capabilities supported by a particular network device. The third goal of this paper is to introduce programming language which use new-defined API to Programmable Abstraction of Datapath (PAD) for different kind of network devices. The presented solution ensures therefore flexibility and adaptability of the new programmable functions to specific requirements of a device. The proposed solution creates a unified way of controlling and configuring a variety of families of network devices from optical switches to x86-based appliances.**

Keywords— SDN; OpenFlow; hardware abstraction; forwarding plane; Programmable Abstraction of Datapath;

I. INTRODUCTION

Open Flow [1] is foreseen as a successful hardware abstraction, however some research projects have already identified several limitations of the current OpenFlow specification, e.g. ALIEN [2][3]. We believe capabilities of modern networking hardware are too heavily restricted by the OpenFlow abstract device model and there is still a need for more elastic and powerful solution. In this paper we would like to address the following limitations of OpenFlow:

- Lack of the switch autonomous capabilities (e.g. MAC learning, NAT port allocation for a new flow) causing heavy usage of OpenFlow *packet-in* messages which affects the scalability of current OpenFlow solutions
- Lack of possibility for generic packet modifications (e.g. adding/removing protocol headers which could be used, i.e., for generating ARP and ICMP responses or implementing new protocols)
- Tight coupling to current network protocols which make it hard to introduce Future Internet revolutionary solutions (e.g. for Named Data Networking)
- More and more network protocols are foreseen to be covered by OpenFlow control which results in a field explosion in OpenFlow specification (tip: big sets of protocol fields are very hard to be packed in the limited sizes of available Ternary Content-Addressable Memory memories - TCAMs).

Taking into account all the above mentioned limitations and drawbacks of available technologies we propose a new approach, which aims to:

i) expose capabilities of network processors through a simple data plane forwarding model applicable for various types of networking hardware with very diverse capabilities (i.e.: programmable network processors, CPUs, optical devices, GEPON, DOCSIS), and

ii) create a library-based interface for managing and controlling data plane protocols and forwarding functions.

Related work. The ideal packet forwarding hardware characteristics have been introduced and explained in [4]. Our solution, as a hardware interface that completely hides details of a real hardware beneath, tries to follow these guidelines. Moreover, the approach proposed in this paper strongly leverages on the work done by research teams working on the P4 language [5] and POF [6] which both provide solutions for datapath programmability. However, the P4 language does not focus on specific details of the datapath forwarding model, additionally, all P4 device configuration must be provided at once, before the device goes to operation. The POF solution becomes less optimal when more protocol options (i.e.: VLAN, MPLS tags, etc) must be handled and for each possible option a new match entry must be added. Both P4 and POF solutions assumes also packet-oriented devices. The ALIEN Hardware

Abstraction Layer (HAL) [7] with its AFA interface aims to support diverse hardware types but is designed only for OpenFlow 1.x protocol. OF-DPA [8] implements the abstraction mechanisms specifically designed for Broadcom Ethernet switches and does not support deep programmability of datapath elements. Fig. 1 presents possible relations between the PAD concept and solutions already available in the research community. P4 or HAL AFA may use the PAD API library for accessing the hardware.

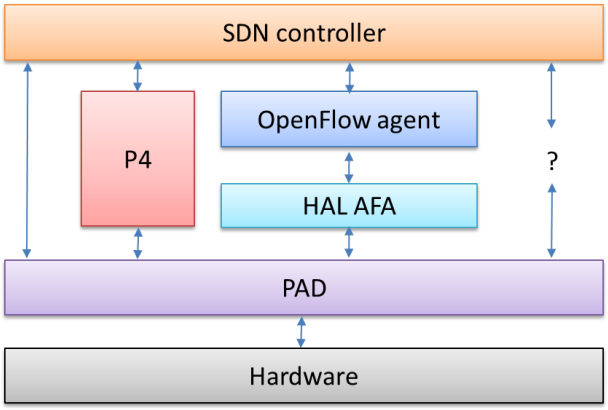

Fig. 1. The Programmable Abstraction of Datapath in in relation with other works in this area

In this paper we introduce Programmable Abstraction of Datapath (PAD), a hardware abstraction (programming model) for data plane elements. The PAD allows programming of datapath behaviour using generic byte operations, defining protocol headers and providing function definitions. Our solution includes also a system for reporting device capabilities in order to provide unified support for diversity of network devices. The PAD is not an extension or generalization of OpenFlow 1.x. The proposal elaborated in this paper derives functionalities exposed by the data plane hardware, not focusing on requirements exposed by the control plane, which continuously evolves, and trying to meet overall objectives of the SDN paradigms.

The remainder of this paper is organized as follows. In Section 2 the abstract model of datapath is described. Section 3 describes the API that provides an interface for managing and controlling hardware hidden behind the abstraction. Section 4 provides an overall description and examples of programming languages used for protocol and function definitions. In Section 5, we present how our solution can be parameterized to abstract datapath elements, e.g. network processors or optical switches.

II. Programmable Abstraction of Datapath

The forwarding abstraction is a model of network device's processing mechanism. The solution proposed in this paper, Programmable Abstraction of Datapath (PAD), comes from the fact that all packet network devices perform similar steps during packet processing: reading packet headers (parsing), making forwarding decisions based on a current configuration (searching), performing necessary actions (modifying and forwarding). This observation has been used as an inspiration for the PAD network device abstraction presented on Fig. 2.

The PAD architecture is composed of several functional components, including ports, search engines, search structures, an execution engine and forwarding functions. Within the PAD, a packet may be processed several times through all these blocks. A packet received from the ingress port is bonded with metadata and passed to the search engine. After the successful search, a packet metadata and search result is passed to the execution engine. A search result contains a function name that will be executed on a packet. At the final stage the packet is passed to the egress port. In most cases, packet processing may run several passes through the PAD using a loopback logical port.

The PAD can be used also for controlling network devices which do not operate on packets (i.e.: optical switches). In this case, the PAD is processing only metadata filled with information, in example, about input port and wavelength of the signal. Matching operations, performed within search engine, are applied on keys composed of metadata fields. Then, the forwarding functions cannot contain packet-oriented instructions but use other (i.e.: optical specific) instructions.

The port in the PAD can be either physical or logical. Physical ports represent physical interfaces of a specific network device. Logical ports are parts of a device abstraction model and are not co-related with any entity on a physical network device (however, particular PAD implementations can use some physical entities to implement this functionality). The loopback logical port is a unidirectional port that interconnects egress and ingress sides of the PAD and allows for multi pass packet processing. A controller's logical port is a bidirectional port that allows for transmission of the processed packet to and from the controller through the northbound interface. Each port (physical or logical) can have a number of sub-ports (e.g. representing different channels on a single optical interface or different SSIDs on a wireless interface).

The search engine is a logical module that performs search operations on search structures. The search key is created from parsed fields from a network packet and metadata (i.e. an ingress port and a sub-port number). The search result is a name of the function defined in the execution engine and packet processing will start with a call of this function.

Each PAD implementation has to support at least one search structure. The number of supported search structures in the PAD model is not restricted. The search structure number 0 is always used for the first pass of the packet processing. Structure numbers for next passes are inserted in a packet metadata and sent together with a packet through the loopback port. In each pass only one search structure can be used. By default, each structure should be a ternary search structure (i.e. support masking of specific bits in a key). Some PAD implementations can additionally support a definition of exact match search structures (i.e. structures without masking possibility). Search structures are defined by their ID number and a structure of a key. The key can be

composed of generic "fields" (e.g. an ingress port number, first 6 bytes and 4 bytes starting from byte 12) or previously

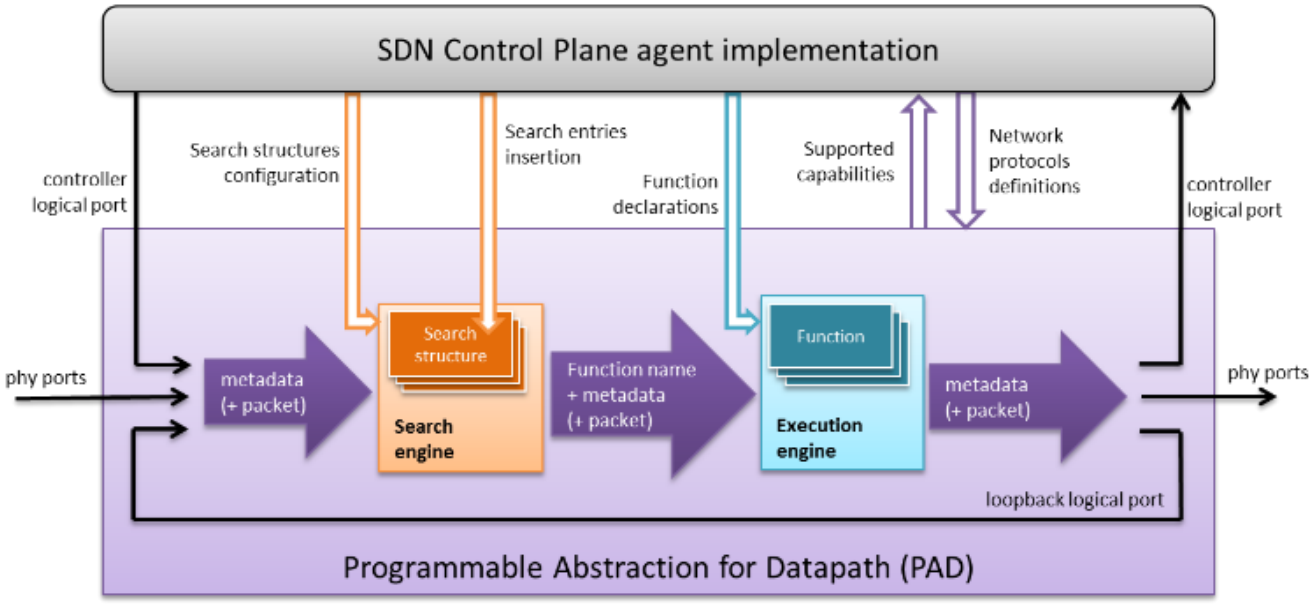

Fig. 2. PAD forwarding element abstraction

defined protocol fields (e.g. an ingress port number, a destination mac address, ethertype and vlan tag, if exists). Both possibilities can be combined in a single structure definition.

The execution engine executes (interprets) functions, which are sets of hardware independent instructions, and translate instructions into hardware-specific actions. Functions declared in execution engine can be called from the body of other functions. The first executed function for a given packet is the one passed from the search structure. A processed frame can be modified here (some devices, such as optical, could not support this feature). In most cases, the processing should finally result in a transmission of a processed data packet to the egress port.

III. THE PAD API

A northbound interface of the PAD allows controlling the network device behaviour. The PAD's northbound interface functionalities can be classified into three functional groups:

Datapath Capabilities - this group of operations allows for information retrieving about capabilities which are supported by a specific hardware. Each PAD instance can implement a certain set of the datapath functionalities. This part of the northbound interface is be used for getting information about search structures limitations and available instruction set before search structures are configured and functions declarations will be installed. More information about datapath capabilities exposed by the PAD could be found in section 5 of the paper.

Datapath Management - this part of the interface allows for managing search structures, functions and defining network protocols. New search structures can be defined with a unique ID and a key description. The management part of this API defines rules of network device operations. The full configuration of a device will require several invocations of appropriate commands. However, the device should not be able to operate with incoherent and incomplete configuration. For this reason, all operations from this part of the API have to be committed in a single, atomic operation to take effect. The PAD implementation accumulates all changes, prepares the new configuration and loads it into the hardware after the commit command is executed with minimal possible interruption on the actual traffic processing. Some management operations, e.g. adding a new function can be theoretically performed without affecting the network device operation. The management part in the API could be extended with additional operations allowing such modifications without stopping the datapath and committing changes. This set of operations, available in the run-time, is optional and supported only by certain PAD implementations.

Datapath Control - This part of the PAD interface allow for adding and removing of entries in defined search structures. In most cases, first two parts of the PAD API are used before the packet processing starts. i.e. the PAD user retrieves device capabilities, configures search structures, uploads functions and then starts the packet processing routines. The control part of the interface, in opposite, is used during the entire time of the device operation, up to many thousands times per second depending on the application.

The PAD API is designed to be very generic and transparently handle network protocol and function definitions which decouple the PAD API with protocol and function specifications methods. The PAD API methods (see

TABLE I.) use textual coding of parameters containing, e.g: a string representation of protocol definitions.

TABLE I. PAD API METHODS FOR ANSI-C

API group	PAD API function	Function description
Capabilities	char* **get_all_capablities**()	Returns a string containing comma-separated list of supported capabilities in the datapath
	char* **get_capability**(char* capability_name)	Returns a string containing a capability value or empty string when no such capability.
	char* **get_basic_instructions**()	Return a string containing comma-separated list of basic platform instructions.
Management	bool **replace_protocols**(char* protocols_spec)	Returns true if network protocols specification where successfully installed in the datapath
	bool **add_protocol**(char* protocol_spec)	Returns true if a network protocols where successfully updated with provided protocol specification
	bool **remove_protocol**(char* protocol_name)	Returns true if a given protocol knowledge was successfully removed from the datapath
	bool **add_structure**(uint8_t id, char* key, uint32_t size)	Returns true if a search structure with a given key schema and total number of entries were allocated within the search engine.
	bool **remove_structure**(uint id) bool **remove_all_structures**()	Returns true if a given search structure (or all structures) were deleted in the search engine.
	bool **add_function**(char* name, char* definition)	Returns true if a function with a given name and definition were successfully added/updated in the execution engine.
	bool **remove_function**(char* name) bool **remove_all_functions**()	Return true if a function with a given name (or all functions) was removed from the execution engine.
	bool **commit_configuration**()	Return true if the PAD configuration is consistent and the datapath was initialized.
Control	uint8 **add_entry**(uint8_t structure_id, uint64_t key, uint64_t mask, char* result)	Return '1' if key, mask and result values were added successfully as a search entry to a given search structure. If key already exist then entry result value is replaced and function returns code '2'. A *result* contains both forwarding function name as all parameters values required by the function.
	bool **remove_entry**(uint8_t structure_id, uint64_t key, uint64_t mask)	Return true if a search entry containing given key and mask were removed from a given search structure.

IV. FORWARDING FUNCTIONS AND NETWORK PROTOCOLS PROGRAMMING

The PAD API requires the use of some kind of forwarding function declarations and protocol definitions programming. These two applications create very different requirements and therefore we propose the use of two different languages.

The programming language is used to define operations performed by the execution engine. Sets of operations are loaded into the program memory as named functions. For each packet, one or more functions are executed.

The primary goal of this language is to allow developers to modify packet in any way including removing existing packet and creating a new one. For this reason the language has to provide wide but specific capabilities that can be easily extended using supported capabilities mechanism.

Key features of this language are:

- Conditional statements and loops
- Fixed-point variables
- Arithmetical and logical operations
- Bitwise operations
- Remove, insert and modify any byte in the packet
- Checksum computation
- Send packet to output port
- Add/delete an entry to/from search structure

In case of implementations that support data plane network protocol definition capability, all defined protocol fields names are also available in the programming language and can be used within search keys and inside functions bodies. All fields that already exist in the packet header can be accessed as regular variables. New empty structure of the protocol header can be inserted into any place in the packet.

The network protocol definition language allows programmers to introduce new data plane network protocols to a network device. Without this feature, the entire packet is seen only as an array of bytes that can be accessed only by its index. The definition of the protocol header consists of fields names, order and sizes as well as information about protocol encapsulation. Protocol encapsulation explains how a given protocol header is linked to other headers, e.g. value '0x86DD' of "Type" field in Ethernet header means that the next header will be IPv6.

This feature allows using specific header fields in search structures or functions definitions regardless of their actual position in the packet. For example, the IP destination address can be used in routing table search structure definition regardless of possibility of VLAN header occurrence that will change the location of the IP header in the packet. In implementations that do not support protocol definition, the programmer needs to consider all possible locations of given field in packet and handle them independently. The protocol definition language can be

defined as a new solution or can be based on existing solutions like NetPDL [9] or P4 [5] languages.

Both languages (i.e.: NetPDL and P4) are still intensively considered regarding capabilities to express a broader scope of data plane protocols headers. However, we would like to present PAD API anyway by an example use shown in LISTING I. This example covers programming booth the search engine and the execution engine with usage of Python wrapper library of the PAD API presented in Table 1. This code snippet presents example implementation of a simple label switching router using generic byte operations. Switching is based on a value of 4 byte long field inserted just after the Ethernet header and announced by the Ethertype value of '0x88b5'. The function call in the line 4 defines a search structure with the key of the length of 6 bytes composed from ethertype value (2 bytes) and 4 following bytes. The string variable defined in the line 8 contains a definition of a simple function that sends a packet to the output port given as a parameter. A value of port parameter is provided by search result. Line 12 adds the previously defined function to the execution engine. The function defined in the line 14 removes the ether type and the tag value before invoking previously defined function 'send_to'. The function defined in the line 21 adds a new tag header to the processed packet and sends them to the output port. The whole new configuration is installed on the datapath with commit command in the line 29. Lines 31, 33 and 35 add entries to the defined search structure that respectively switches packets with the tag value '0x17' to the port 7, switches packets with the tag value '0x18' to the port 8 and removes tags from the packet with the tag value '0x11' before sending it to the port 1.

LISTING I. EXAMPLE OF USE of PAD API in Python

```
 1: from pad import add_structure, add_function,
 2:                 add_entry, commit_configuration
 3:
 4: add_structure(id=0,
 5:     key="""2 bytes from byte[12].bit[0],
 6:            4 bytes from byte[14].bit[0]""")
 7:
 8: function = """
 9:     send_to(port){
10:         send_to_physical(port);
11:     }"""
12: add_function(definition=function)
13:
14: function = """
15:     decapsulate_and_send(port){
16:         remove(from=byte[12].bit[0], length=6B);
17:         send_to(port);
18:     }"""
19: add_function(definition=function)
20:
21: function = """
22:     encapsulate_and_send(tag, port){
23:         insert(after=byte[12].bit[0], value=0x88b5);
24:         insert(after=byte[14].bit[0], value=tag);
25:         send_to(port);
26:     }"""
27: add_function(definition=function)
28:
29: commit_configuration()
30:
31: add_entry(structure_id=0, key=0x88b500000017,
32:           mask=0xffffffffffff, result="send_to(7)")
33: add_entry(structure_id=0, key=0x88b500000018,
34:           mask=0xffffffffffff, result="send_to(8)")
35: add_entry(structure_id=0, key=0x88b500000011,
36:           mask=0xfffffffff,
37:           result="decapsulate_and_send(1)")
```

V. DEVICE CAPABILITES

Different network devices support different capabilities. Not all of hardware platforms allow manipulating forwarded frames and packets. In particular, optical devices allow only circuit switching in the forwarding plane. Narrow set of network devices mostly based on network processors and programmable entities like FPGAs provide an access to the hardware datapath through APIs. Heterogeneity of network devices and forwarding paradigms causes problems with a definition of the unified abstraction for all types of network devices [10]. The ALIEN project introduces Hardware Abstraction Layer [2][7] which hides the complexity of various network devices and exposes an OpenFlow endpoint for devices that don't support this protocol natively. However, this abstraction is oriented on OpenFlow only, with some additional limitations causing non-use of heterogeneity of devices' capabilities.

The PAD which exposes capabilities of different hardware platforms makes the concept of hardware abstraction more general and unified. The possibility of using only one chosen part of PAD functionality ensures elasticity of the presented solution. Each physical device is expected to support only a part of PAD's functionalities that is appropriate for the device. A well-defined system of supported capabilities will allow implementing the PAD on top of optical switches as well as network processor appliances without limiting their capabilities.

The key parameters included in capabilities definition:

- Maximal number of search structures
- Maximal length of a key in search structure
- Support for exact matches
- Supported instruction sets
- Support for protocol definitions

For example an optical switch can be presented by the following capabilities (because of the nature of the traffic forwarding at the optical level):

- Only metadata in search structure key (without direct access to frame)
- Sending the frame to port (without frame modification)
- Dropping the frame

Programmable Ethernet switches with full access to the frame support:

- Compound keys in search structure enable matching to different frame header fields as well as frame

metadata with counters, ingress port identification etc.

- Sending the frame to port
- Modifications of the frame
- Dropping the frame

In the case of closed platforms or platforms with limited access to datapath, it is possible to implement the PAD model which uses an available management interface to the device only (e.g. CLI, SNMP). Because of different capabilities of network devices not all functions in execution engine will be installed in the PAD as well as entries in search structures will be adjusted to supported device capabilities.

VI. SUMMARY AND NEXT STEPS

The PAD internal architecture presented in section 2 should not be considered as a complete set of software modules for implementation on each hardware platform. Each implementation will provide functional equivalent of presented abstraction using the software architecture suitable for the given hardware platform. The presented solution can be seen as a primary interface for all interactions with datapath of hardware devices. As an open hardware abstraction layer that can be used by local control plane processes as well as by remote network controllers using middleware protocols.

OpenFlow, the most popular SDN protocol, can be implemented on top of the PAD as a middleware for compatibility with existing controllers. Such device configuration will present to the controller only functionalities supported by OF, but, by using standardized hardware interface, will be open for replacing OpenFlow implementation with a new one, overcoming current limitation of the OF protocol.

The locally only available PAD API requires a middleware protocol to be used by a remote network controller, however it can be also used by a local control software. In such configuration, the PAD can be used as a hardware abstraction layer for implementation of legacy network protocols like STP or OSPF.

The PAD has been designed with ease of use in mind, but still is relatively easy to implement on most of hardware architectures. The PAD architecture can be directly converted into software modules which makes implementation straightforward on C language programmed network processor platforms like Cavium Octeon, Broadcom XLS/XLR/XLP or x86 supported by Intel DPDK. On EZchip NP based devices the code generated for parsers can be deployed in TOPparse and TOPsearch and can be used for all searches. Compiled actions can be deployed on TOPmodify and also partially in TOPresolve if additional instructions are needed.

Some considerations have been made in this paper regarding implementation of the PAD prototype model but there are still open issues that should be solved. A list of hardware capabilities for all different network platforms must be gathered. Regarding the methods of actions definitions, we must decide if we would like to enhance our current design or switch to other propositions like P4 language. We must also address very important aspects of both packet processing and control operation performance which could different for particular platform implementation. Performance of function executions during packet processing should be better with pre-compilation of the instructions than on-fly interpretation. Performance of compiled implementation will heavily depend of quality of the compiler. From control plane point of view, most of time consuming configuration tasks will be performed once after device start. Adding and deleting entries will be implemented as a population of defined data structures which should be as fast as in the case of OpenFlow 1.x. Adding new functions (if implementation supports this feature in runtime) will be fast with interpreted implementations and will take some time for compilation in compiled implementations.

The PAD architecture proposed in this paper enables generic abstraction of different kind of network devices. The PAD model is not restricted only to Ethernet-based protocols. An opportunity to change the network device behaviour (also on the fly) based on a well-defined "network program" is a flexible solution which allows changing frames, packets or datagrams handling on the fly. The ability of fine-grained definition of the forwarding behaviour of network equipment gives new opportunities towards the SDN concept.

ACKNOWLEDGMENTS

This work has been conducted within the framework of the EU-FP7 ALIEN project [3], which is partially funded by the European Commission under grant agreement no. 317880.

REFERENCES

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, April 2008.

[2] (2012) ALIEN project, "HAL whitepaper" [Online]: http://www.fp7-alien.eu/files/deliverables/ALIEN-HAL-whitepaper.pdf.

[3] (2014), FP7 ALIEN project, website, http://www.fp7-alien.eu

[4] M. Casado et al., "Rethinking Packet Forwarding Hardware", ACM SIGCOMM HotNets Workshop, Nov. 2008.

[5] Pat Bosshart at al. "Programming Protocol-Independent Packet Processors", arXiv:1312.1719v3 preprint, May 2014.

[6] H. Song, "Protocol-oblivious forwarding: Unleash the power of SDN through a future-proof forwarding plane", ACM SIGCOMM HotSDN Workshop, Aug. 2013.

[7] Ł. Ogrodowczyk et al., "Hardware Abstraction Layer for non-OpenFlow capable devices", Terena Conference, 2014, Dublin; https://tnc2014.terena.org/getfile/1047.

[8] (2014) Broadcom, "OpenFlow Data Plane Abstraction (OF-DPA)" [Online]; http://www.broadcom.com/collateral/pb/OF-DPA-PB100-R.pdf.

[9] M. Baldi F. Risso, "NetPDL: An Extensible XML Based Language for Packet Header Description", Computer Networks, vol. 50, no. 5, pp. 688-706, 2006.

[10] Deliverable D3.1 from ALIEN PF7 project, "Hardware platforms and switching constraints"; http://fp7-alien.eu/files/deliverables/D3.1-ALIEN-final.pdf.