Unified POF Programming for Diversified SDN Data Plane Devices

Haoyu Song, Jun Gong, Hongfei Chen, Justin Dustzadeh

Huawei Technologies
Santa Clara, CA, USA, 95124
email: {haoyu.song, jun.gong, hongfei.chen, justin.dustzadeh}@huawei.com

***Abstract*—Software Defined Networking (SDN) will ultimately evolve to be able to program the network devices with customized forwarding applications. The ability to uniformly program heterogeneous forwarding elements built with different chips is desirable. In this paper, we discuss a data plane programming framework suitable for a flexible and protocol-oblivious data plane and show how OpenFlow can evolve to provide a generic interface for platform-independent programming and platform-specific compiling. We also show how an abstract instruction set can play a pivotal role to support different programming styles which map to different forwarding chip architectures. As an example, we compare the compiler-mode and interpreter-mode implementations for a Network Processing Unit (NPU) based forwarding element and conclude that the compiler-mode implementation can achieve a performance similar to that of a conventional non-SDN implementation. Built upon our Protocol-Oblivious Forwarding (POF) vision, this work presents our continuous efforts to complete the ecosystem and pave the SDN evolving path. The programming framework could be considered as a proposal for the OpenFlow 2.0 standard.**

Keywords–SDN; OpenFlow; POF; data plane; programming.

I. INTRODUCTION

It has been envisioned that in SDN the network intelligence should be moved to software as much as possible in order to support agile, flexible, and low-cost network service deployments. Programmable Forwarding Elements (FE) are essential to enable this vision and represent a big leap from the current network device application paradigm. These FEs will be shipped as white-box just like bare-metal servers without fixed functions or pre-installed applications. User can program the device through a standard-based open interface. Therefore, the future SDN operation can be modeled as follows. First, the user determines the entire forwarding protocols and behavior through device-level programming. Note that this is done through high level programming over high level device abstractions. After this step, the white-box is equipped with customized functions tailored for operator needs. Then, the operator applies runtime control to operate these devices through network-level and service-level programming. Any third party can produce software to program and configure the programmable FEs. Any third party can also produce network-level application software which taps into the customized FEs to offer various network services. Depending on the actual use cases, the role of device programmer, service programmer, and network user can be overlapped or independent. The advantages of this network operation model are obvious. Network applications can be programmed on-the-fly and deployed in real time. Service innovation is never so easy and accessible before. Moreover, the system time-to-market can be significantly reduced and the life cycle of FEs greatly extended.

In the arena of programmable data plane devices, Central Processing Unit (CPU) and NPU-based FEs are clearly qualified candidates, but so far, there is lack of an open and standard interface for forwarding application programming on these devices. In most cases, these devices are still programmed by vendors and shipped to users in the form of virtual or physical appliance. Some open-source soft switches, such as Open Virtual Switch (OVS) [1], allow user to modify its behavior but apparently this process is labor intensive and target dependent. Hence, the application model of these FEs is not so much different from those built with fixed-function switch chips based on Application Specific Integrated Circuit (ASIC).

While not fully programmable, ASIC chips are usually configurable to some extent and able to handle most of popular Data Center (DC) switch applications. ASIC-based FEs can be considered to have pre-installed packages or standard library functions. With certain negotiation process, such as Table Type Pattern (TTP) Negotiable Datapath Model (NDM) [2], ASIC-based FEs can still be controlled under the same SDN framework, as if they were programmed by the controller.

Recently, a new breed of SDN-optimized programmable chip is investigated [3]–[5]. These chips aim to support flexible SDN application programming without compromising performance. If succeed, this new contender will further accelerate the SDN transformation. It is worth to mention that Field Programmable Gate Array (FPGA), a reconfigurable chip by nature, can also potentially play a similar role with proper design-flow refactoring. SDNet [6] represents such an effort.

For the foreseeable future, diverse FEs built with different chips will coexist in various network segments. As such, it is critical to have a unified framework, not only to control and program these FEs, but also to hide the heterogeneous substrate architecture and present a unified programming interface to SDN controller and applications. We position OpenFlow [7] as the center pillar of this framework. OpenFlow abstracts the SDN data path as a pipeline of tables and actions. This model is arguably the easiest way to map the forwarding functions to any target FEs. However, further investigation and work are needed to address some of the challenges with the current approach (e.g., fixed protocol support and stateless data path) [5], [8].

We believe the next generation of OpenFlow (e.g., OpenFlow 2.0) should offer the following capabilities: (1) Allow a protocol-oblivious data plane so that no packet format and network behavior need to be hard-coded in FEs. This capability is important to maximize SDN's flexibility and extensibility. (2) Allow an FE-agnostic SDN controller so that the data plane abstraction can help isolate the controller from the FE im-

 ISBN: 978-1-61208-404-6

plementation details. This capability is important to minimize SDN's efforts to program heterogeneous substrate platforms. (3) Allow coexistence of static programming through the use of packages or library functions and dynamic runtime programming/reconfiguration through the use of flow instructions. This capability extends the usability of diversified FEs and can offer the needed flexibility to satisfy some special SDN use-case requirements. While audacious, these goals represent the right direction for OpenFlow evolution. In this paper, we present an OpenFlow-based SDN FE programming framework and provide our experience on realizing it.

The remainder of the paper is structured as follows: Section II describes the proposed programming framework; Section III provides a case study on an NPU-based platform; Section IV discusses the related work; and Section V concludes the paper.

II. Unified Programming Framework

The unified SDN data-plane FE programming framework is depicted in Figure 1. The center pillar of this framework is the standardized OpenFlow 2.0 interface, which provides a set of generic instructions, as well as other data-plane provision and monitoring mechanisms. The interface provides a decoupling point between the control plane and the data plane. It is versatile and protocol/platform-agnostic [8].

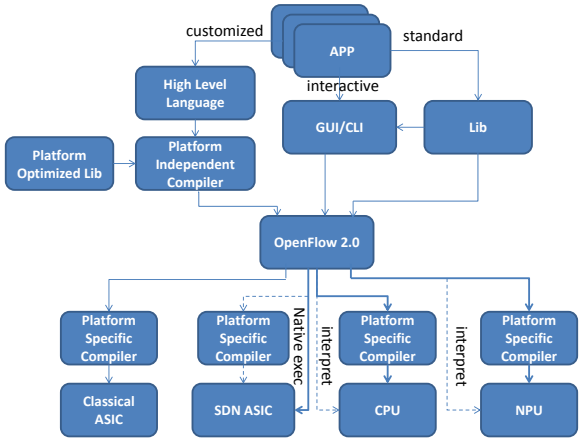

Figure 1: SDN Data Plane Programming Framework

A. Intermediate OpenFlow Interface

OpenFlow is pivotal for the data plane programmability. The key to a successful design of such a programming interface is to make it work at the right abstraction levels. In particular, the interface should not be tied to a particular FE architecture. Instead, it should allow programs to be easily mapped to any target while allowing specific optimizations to fully exploit the target-specific capability.

To this end, we propose a simple yet generic abstract forwarding model, as shown in Figure 2. In this model, the "In Ports" and "Out Ports" are the source and sink of a packet under processing. The port can be either physical or logical. It can be anything that is out of the scope of the processing directly programmed by user. For example, the controller, a service card, some black-box network functions, and packet recirculation can all be abstracted as ports. This abstraction guarantees only a single packet is in the processing pipeline at a time and the packet is uniquely identified by its input port.

The packet processing is abstracted as a sequence of search tables and the corresponding actions triggered by table lookup results. Note that this view is also roughly held by OpenFlow but we purify it to an extreme. The table can match on any designated packet field or metadata, and the matching result points to an action which comprises a block of instructions.

The abstract forwarding model can easily describe any packet processing tasks. For example, to map to some target hardware with a front-end parser, the very initial table is defined as a port table and the following action contains instructions which parse packets and extract header fields. The number of instructions allowed in an action depends on the target hardware and the performance constraints. A target-specific compiler can also find the parallelism opportunities within an action and takes advantage of it.

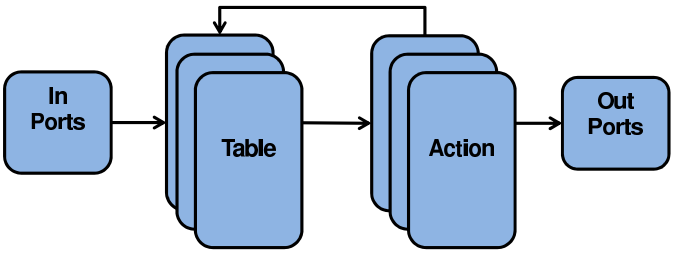

Figure 2: Abstract Forwarding Model

The core of our proposed OpenFlow 2.0 interface is a set of generic "flow" instructions (i.e., POF Flow Instruction Set (FIS)) [9], which are used to program the actions. These instructions function as the intermediate language between the platform-independent programming environment and each individual target platform. The instructions are grouped and summarized as follows:

- *Packet/Metadata editing:* set field, add field, delete field, math/logic operations on field in packet or metadata
- *Flow Metadata manipulation:* read, write global data (e.g., counters, meters) for stateful operations
- *Algorithm/Function procedure:* checksum, hashing, random number generating, etc. Extensible to include other standalone black-box functions
- *Table access:* go to table (non return), search table (return to calling instruction) with keys extracted from packet and metadata
- *Output:* to physical/virtual/logical port, with sampled/data-path generated packets, or original/modified/mirrored/cloned packets
- *Jump/Branching:* conditional and unconditional, absolute and relative
- *Active data path:* insert/delete/modify flow entry, insert/delete flow table
- *Event:* timers

These instructions operate on the objects such as packet data, meta data, and flow tables. Note that the instruction set we defined by no means complete. More instructions can be included in the future as the scope of packet processing is extended to cover tasks such as queuing and scheduling.

 ISBN: 978-1-61208-404-6

In addition to making the flow instructions protocol-oblivious, we propose other new features to enhance the programmability and to enable performance optimization. One notable addition is the ability to abstract the actions associated with each flow entry as a piece of program. This provides several advantages. For example, it allows decoupling match keys and actions. The actions for flow entries, in form of instruction blocks, can be downloaded to FEs separately from flow entry installations. When each instruction block is assigned a unique identifier (ID), the flow entry only needs to include a block ID to infer the associated actions. By doing this, not only different flow entries can share the same instruction block while an instruction block is only downloaded and stored once, but also there would theoretically be no limit on how many instructions one flow entry can execute. By using the *goto-table* instruction within an instruction block, the table traversing order (i.e., the processing flow) can be dynamically changed. Instruction block update is also easy: one can simply load a new instruction block, update the block ID in affected flow entries, and then revoke the old instruction block if it is not needed anymore.

To facilitate instruction block sharing and at the same time enable differentiated flow treatment, we augment the flow entry with a parameter field. This field can be leveraged by application developers to define any parameters used by the associated instruction block. For example, in an egress table, when all the entries execute an output action, they may have different target output ports. While the output action is coded in an instruction block and shared by all the flow entries, the output port number is stored in the parameter field of each flow entry. This is just an overly simplified example. In reality, this mechanism is efficient in code space reduction.

We also abstract the globally-shared memory resource as a flow metadata pool. Flow metadata can be shared by flow entries to store statistics (i.e., counters) or any other information such as flow states. This is another enhancement on top of the existing packet metadata mechanism which is only dedicated to each packet. In particular, the expressivity of flow metadata enables stateful data-plane programming.

B. Programming over OpenFlow Interface

Above the OpenFlow interface, any network forwarding application needs to be converted to the standard OpenFlow configuration commands and instruction blocks first. There are three ways to do it. First, it would be handy to use some high-level language to program network applications on devices. The high-level language provides another layer of abstraction that supports modularity and composition [10]. With the help of a high-level language, developers can focus on application functions rather than dealing with particular FE architecture and conducting error-prone table and flow manipulations. Some SDN programming languages have been proposed in literature [11], [12]. However, they are more focused on network-wide policy deployment on FEs built with fixed-function chips. Recently, a device-level programming language called P4 was proposed [13]. Some latest NPU chips are made C-programmable [14], [15], albeit only accessible by device developers. No matter which language is picked, a new compiler needs to be developed for sure. But, using a popular language can shorten the learning curve and increase the productivity. We are exploring the possibility of using C and Java as our choice of high level language. However, this is still an open and active research area. Until we thoroughly fathom the feasibility, we do not exclude other possibilities.

Although programming in a high-level language is meant to be forwarding-platform-independent, we realize that in the near future, many different forwarding architectures will coexist. For example, some chips (notably ASIC-based chips) have a physical front-end packet parser which parses packets in a centralized way but some other chips (notably NPU-based chips), for performance reason, prefer incremental packet parsing where packets are parsed layer by layer along the packet processing pipeline. Moreover, each kind of chip may have its own feature extensions, hardware-accelerated modules, and other nuances in hardware resource provisioning. Without discerning these differences, a generic program would pose significant challenges to the complier, which may lead to poor performance or even worse, failure to compile at all. Therefore, the application program should follow some programming style constraint upfront and may include some preprocessor directives to guide the compiling process. The key point is that the language itself must be general enough. The platform-independent compiler compiles the application programs by calling the platform-optimized library and generates an Intermediate Representative (IR), which will be passed down to FEs through the OpenFlow interface. This is not a perfect solution from a purist's perspective. However, as long as the FE chips do not converge to a single architecture, we have to live with it. The good thing is, if in the future the chip architectures do converge, the design flow and interface do not need to change.

Another method is to directly use Graphical User Interface (GUI)/Command Line Interface (CLI) for interactive and dynamic data plane programming at runtime. This could be considered similar to low-level programming in assembly language. Although it needs to handle flow level details, this method is fast and can fully explore the FE flexibility. The GUI/CLI can be used to handle fast runtime reconfigurations and can also be used to directly download compiled applications to FEs. We have implemented an open-source GUI to support this programming method [16].

At last, there are many prevailing network applications and forwarding processes today. For example, the basic Layer 2 (L2) switching and Layer 3 (L3) Internet Protocol (IP) forwarding are still widely used. It would be counterproductive to try to develop them again and again. Also, some applications on some particular target platforms may have been deeply optimized to achieve the best possible performance. It would be difficult for inexperienced developers to implement these applications with a similar performance. Therefore, pre-compiled applications can be provided in a library by any third party and directly used to program the network. Conceptually, this is in line with the TTP developed by Open Networking Foundation (ONF) Forwarding Abstraction Working Group (FAWG) [2]. Once the specifications of these library applications are standardized or publicized, any third party can develop and release them. Users can also maintain their private library and download the program through GUI or CLI.

Note that these programming approaches are not mutually exclusive. In other words, an application could be implemented through the simultaneous use of more than one approach. In a typical scenario, the basic forwarding process is either customized by using the high-level language or taken from a standard library application, and then GUI/CLI is used for library application download, dynamic runtime updates, and

 ISBN: 978-1-61208-404-6

interactive monitoring.

C. Programming Diversified Platforms

Once the program in the form of standard IR is conveyed to the FEs through OpenFlow messages, the FEs may have its own platform-dependent compiler which compiles or maps the program to its local structures. Note that this platform-dependent compiling process can also happen in controller. In this case, the burden on FEs is alleviated but the controller would need to retain extra knowledge about the target platform. The pros and cons of both options are still open to debate, but we believe our choice represents a clean architectural cut and is better for a coherent OpenFlow interface which can seamlessly support both configuration and operation. We roughly categorize FEs into four groups based on the type of main forwarding chips on them.

1) Conventional ASIC-based: Conventional ASICs for FEs typically have a fixed feature set and are not openly programmable. However, since they are designed to handle classical forwarding scenarios at high performance, they are still usable in SDN but in a more restrictive way. In this case, the standard library applications are the most suitable way to "program" the FEs. Some ASICs are configurable and can switch between different modes to support different applications. In this case, customized programming is not impossible but needs to be applied in a highly-disciplined way to ensure compatibility.

2) SDN ASIC-based: Recent research has started to pay more attention to SDN-optimized chips [3], [17]. Some companies are planing or have started to develop chips to better support flexible network application programming [4]. We can also put FPGA, if properly designed, into this category. These chips have embedded programmable capability for general packet handling but are also heavily populated with hardware-accelerated modules to handle common network functions for high performance. For these chips, it is feasible to use any kind of programming method. Due to the architectural limitations (e.g., hardware pipeline), low level interactive programming may not be well supported in these FEs. Therefore, the customized programming and application installation are preferred. A target-specific compiler is needed to compile the IR into the chip's local structure.

The compiler, no matter how well-designed, may cause some performance loss due to the extra level of indirection. When the OpenFlow 2.0 is standardized, it is conceivable that in the future we could even design a chip that can natively execute the POF-FIS instructions without even needing a compiler in data plane.

3) CPU-based: CPU is no doubt the most flexible platform. Albeit having lower performance compared with the other platforms, it can easily support any programming method. Software-based virtual switches (e.g., OVS) are widely used in data centers. The switch implementation in CPU can basically run in two different modes: compiler mode and interpreter mode. The former compiles an application in IR into machine binary code (akin to the customized programming approach) and the latter requires the forwarding plane to directly interpret and execute OpenFlow instructions dynamically (akin to the interactive programming approach). The interpreter mode is more straightforward to implement and allows more flexible usage of the switch. The open source soft switch presented in [16] works in interpreter mode. It is unclear to us which mode has higher performance. We are working on a compiler-mode implementation based on x86 platform which targets on OVS.

4) NPU-based: NPUs are software programmable chips. They are designed specifically for network applications. An NPU typically contains multiple processing cores to enhance the parallel processing capability. NPUs can be broadly categorized into two types: pipeline and Run-To-Completion (RTC).

A representative pipeline NPU is EZchip's NP family chip [18]. In a pipeline NPU, each stage processor only handles a portion of packet processing tasks. Although the pipeline NPU's architecture seems to match OpenFlow's processing pipeline model, in reality it is not easy to perfectly map the two pipelines together because OpenFlow's pipeline is function-oriented and NPU's pipeline is performance-oriented. The compiler needs to carefully craft the job partition to balance the load of pipeline stages.

In an RTC NPU, each processor core is responsible for the entire processing of a packet. This architecture maximizes the programming flexibility, which is similar to CPUs. However, it has limited code space per core and needs to share resources (e.g., memory) among cores. The code space constraint requires the code size to be compact enough in order to accommodate the whole processing procedure (e.g., we cannot afford to repeat the storage of the same set of actions for every flow in a large flow table). The resource sharing constraint requires both the number of memory accesses and the transaction size per memory access to be minimized in order to meet the performance target. Fortunately, the new features we proposed for the OpenFlow 2.0 interface allow software developers to program efficiently with these constraints in mind.

NPU-based FEs can also be programmed in compiler mode or interpreter mode. In the next section, we discuss the implementations of both modes on an NPU-based FE and compare their performance.

III. NPU-BASED CASE STUDY

The NPU-based FE prototype works on Huawei's NE-5000 core router platform. The line card we used has an in-house designed 40G NPU and each half slot interface card has eight 1GbE optical interfaces. The multi-core NPU runs in RTC mode.

A. Forwarding Programming in C

To support high level data plane programming, we model three entities: Metadata, Table, and Packet. The program simply manipulates these three entities and forwards the resulting packets based on the table lookup results. For our NPU, the three entities are all realized in registers. Metadata is used to hold the packet metadata which is represented as a customized structure; Table is the associated data of flow entries loaded from table matches, which is also represented as a customized structure; Packet is typically the packet header under process which is described in another structure.

Figure 3 shows the structures of Metadata, Table, and Packet for an L3 forwarding application. A piece of program that processes a packet is shown in Figure 4. It combines the IP address and the Virtual Private Network (VPN) ID as a new key to conducts another table lookup.

Once the packet processing flow is described in C, it is straightforward to compile the program into IR, which include protocol parsing rules, table specifications, and flow actions.

 ISBN: 978-1-61208-404-6

```
struct Metadata_L3 {
    uint8 L3Stake; //L3 Offset
    uint16 VpnID; //VPN ID
    uint16 RealLength; //Packet Length
    uint16 SqID; //QOS Queue ID
};
struct Table_Portinfo {
    uint16 VpnID; //VPN ID
    uint16 SqID; //QOS Queue ID
};
struct IPV4_HEADER_S {
    uint4 Version;
    uint4 HeaderLength;
    union {
        uint8   TOS;
        uint6   DSCP;
        uint3   Precedence;
    };
    uint16 TotalLength;
    uint16 FragReAssemID;
    IPV4_FRAG_HWORD_S FragHWord;
    IPV4_TTL_PROT_HWORD_S TtlProtWord;
    uint16  Checksum;
    uint32  SIP;
    uint32  DIP;
};
```

Figure 3: Structures for L3 Forwarding

```
(Metadata_L3 *) p_metadata;
(Table_Portinfo *) p_table;
p_metatada->VpnID = p_table->VpnID;
p_ipheader = p_packet + 14;
Goto_Table(TableID, p_metadata->VpnID, p_ipheader->DIP);
```

Figure 4: Code Example

Although the programming style appears to be platform independent, the Goto_Table library function could be specific for each different forwarding platform in the above example. To infer the different platform implementation to the compiler, an NPU-specific proprietary library is included.

B. Interpreter Mode FE Implementation

In interpreter mode, each POF-FIS instruction in a flow action (i.e., an instruction block) corresponds to a piece of code written in NPU microcode which realizes the instruction's function. The code translation is straightforward. However, due to the flexibility embedded in the POF-FIS instructions, the efficiency of the microcode is problematic.

For example, the Goto_Table instruction may lead to a complex microcode processing flow. First, it needs to read the corresponding table information and initialize a buffer to hold the search key, then it enters a loop to construct the search key piece by piece depending on the number of header fields involved in the instruction. Each iteration of the loop contains many steps. It needs to locate the target field using the offset and length information, copy the field into the key buffer, and mask the field. This process requires a lot of pointer shift, data move, and other logic operations. Finally, the search key is sent to the target flow table and the thread is hung up to wait for the lookup result.

The inefficiency comes from three sources: (a) the microcode instruction count, (b) the number of thread switch, and (c) the bandwidth of loading flow table entries. The microcode instruction count is determined by the microcode instruction set and the complexity of POF-FIS instructions. The thread switch is caused by the loops that force to break processing pipelines, as well as the latency for table lookups. Each table lookup will return an instruction block. If parameters are directly carried within instructions, the bandwidth of loading such instruction blocks are considerably expanded. As a result, the throughput suffers. The last inefficiency can be addressed by allowing the flow entry to carry the parameters but this is not enabled in our prototype yet.

In general, the interpreter mode implementation is suitable for the interactive programming approach in which the data path is fluid and can be constantly changed. While this mode is less likely to be widely used in production networks, it is interesting in experimental and research environments for quick design verification.

C. Compiler Mode FE Implementation

In compiler mode, the application is considered a whole and a relatively static entity. This allows the compiling process to simplify the microcode. Since there are a set of registers $R0 \sim Rn$ in NPU, the compiler can resolve the pointer offsets and directly map the data into registers. This eliminates the need of pointer manipulations in microcode. The compiler also handles the length evaluation and directly translates that into assignment statement. These can help to reduce the microcode instruction count by more than 50%.

The compiler mode implementation can easily take advantages of the flow parameter mechanism which reduces the instruction block size. This lowers the bandwidth requirement for memory access and further boosts the throughput and latency performance.

D. Performance Evaluation

The packet forwarding performance in NPU is evaluated by throughput (R) and packet latency (L). We know that $R = c * f/i$ and $L = t/R$ in which c is the number of processing cores, f is core frequency, i is microcode instruction count per core, and t is the number of threads. Given an NPU, c and f are fixed, so the performance is mainly determined by i and t. Reducing table lookup latency and memory access bandwidth have direct impact on t. Table I compares the performance of different Goto_Table implementations (n is the number of match fields in the search key).

TABLE I: *Goto_Table* Performance Comparison

	instr. count	# thread switch
Interpreter Mode	$37 + 33n$	$7 + 3n$
Compiler Mode	13+n	1

Table II summarizes the performance comparison for basic IPv4 forwarding. The conventional non-SDN implementation is used as a benchmark, which has exactly the same function as the SDN-based implementations. The conventional implementation can fully exploit the hardware features (e.g., protocol parsers) and the microcode is deeply optimized.

Through extensive experiments, we found that the compiler-mode implementation performs consistently better than the interpreter-mode implementation. For a typical IP forwarding process in routers, the compiler-mode implementation

 ISBN: 978-1-61208-404-6

TABLE II: Performance Comparison for *IPv4* Forwarding

	non-SDN	Interpreter	Compiler
instr. count	496	1089	550
# thread switch	94	146	74
thruput (Mpps)	77.5	35.3	69.8
latency (cycle)	4468	6361	4022

needs 57% less microcode instructions than the interpreter-mode implementation. Compared with the conventional implementation, the compiler-mode implementation is just 11% worse. With the same number of micro cores, a compiler-mode implementation can easily double the throughput of an interpreter-mode implementation.

IV. Related Work

This paper is concerned with the SDN data plane programming issues. To put it in context, interested readers can refer to Kreutz et al. [5] for an up-to-date comprehensive survey of SDN research and practice.

P4 language is based on an abstract forwarding model [13]. The use of *P4* is akin to our customized programming approach. For an application, it defines the header parse graph and the switch control program. The control program basically describes the tables, the action set supported by each table, and the table dependencies. The model also needs a platform-dependent compiler to map the configuration to each specific target switch. After configuration, the controller can then populate the tables with actual flow entries at run time.

Open Compute Project (OCP) networking project advocates open switches with open-programming environments [19]. Quite a few open switch specifications and open-source softwares have been released since the project debut in 2013. However, at its current stage this project still falls short of SDN support: (1) It focuses on programming in an open Linux-based Network Operating System (NOS) environment for each individual switch but not in a centralized SDN programming environment; (2) The current open switch specifications heavily rely on existing ASIC-based chips and Software Development Kit (SDK)/Application Programming Interface (API) provided by chip vendors. The programming flexibility is limited by the chip architecture and the degree of openness the chip vendors would like to offer. We believe a truly open switch also means open silicon chips or at least a universal and complete API. The project might evolve towards a similar direction as we proposed.

V. Conclusion and Future Work

We believe it is plausible to assume that the next generation SDN will require total programmability over an open data plane. An FE could be programmed as easily as a bare-metal server can be programmed today. However, the diversified chips used to build the FEs today and in the foreseeable future are far from a convergence. This poses a challenge for the desired uniform and coherent SDN programming experience. Until we solve this problem, we cannot claim a vertical-decoupling of the SDN layered architecture is fully achieved. With the current SDN approach, it could become difficult to build an efficient ecosystem in which players would work at different layers independently.

In this paper, we presented our initial exploration and experience on this hard problem. We propose a programming framework which centers on the next-generation OpenFlow interface, targets various FEs, and supports different programming approaches. In particular, we experiment on an NPU-based platform and show that the complier-mode implementation is superior to the interpreter-mode implementation in terms of performance, although interpreter mode implementation offers much better runtime flexibility.

Our future work includes completing the proposed SDN programming framework by implementing the missing pieces in Figure 1 (e.g., platform-dependent compilers for other FE platforms) and demonstrating real-world SDN applications through the full programming process. We are also working on extending OVS to support POF and making it runnable in Mininet environment, so the idea is more accessible to the research community. This programming framework can be considered as a proposal for the OpenFlow 2.0 standard.

References

[1] Open vSwitch, http://openvswitch.org/ [retrieved: March, 2015] .

[2] ONF Forwarding Abstraction Working Group (FAWG), https://www.opennetworking.org/working-groups/forwarding-abstractions [retrieved: March, 2015].

[3] Pat Bosshart et al., "Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN," in *Proceedings of the ACM SIGCOMM*, 2013, pp. 99-110.

[4] White Box Week: Chip Startups Take Aim at Broadcom, https://www.sdncentral.com/news/white-box-week-chip-startups-take-aim-broadcom/2013/11/ [retrieved: March, 2015] .

[5] Diego Kreutz and Fernando Ramos and Paulo Esteves Verissimo and Christian Esteve Rothenberg and Siamak Azodolmolky and Steve Uhlig, "Software-Defined Networking: A Comprehensive Survey," *Proceedings of the IEEE*, January 2015, pp 14-76.

[6] Software Defined Specification Environment for Networking, http://www.xilinx.com/applications/wired-communications/sdnet.html [retrieved: March, 2015] .

[7] Nick McKeown et al., "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Communication Review*, vol. 38, April 2008, pp. 69-74.

[8] Haoyu Song, "Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane," in *ACM SIGCOMM HotSDN Workshop*, 2013, pp. 127-132.

[9] Jingzhou Yu and Xiaozhong Wang and Jian Song and Yuanming Zheng and Haoyu Song, "Forwarding Programming in Protocol-Oblivious Instruction Set," in *IEEE ICNP CoolSDN Worshop*, 2014, pp. 577-582.

[10] Nate Foster et al., "Languages for Software Defined Networks," *IEEE Communication Magazine*, Feburary 2013, pp. 128-134.

[11] ——, "Frenetic: A Network Programming Language," in *ACM SIGPLAN ICFP*, 2011, pp. 279-291.

[12] Andreas Voellmy and Paul Hudak, "Nettle: Taking the Sting Out of Programming Network Routers," in *PADL*, 2011, pp. 235-249.

[13] Pat Bosshart et al., "P4: Programming Protocol Independent Packet Processors," *Computer Communication Review*, 2014, pp. 87-95.

[14] EZchip NPS, http://www.ezchip.com/ [retrieved: March, 2015].

[15] Netronome Flow Processor, http://www.netronome.com/ [retrieved: March, 2015].

[16] Protocol Oblivious Forwarding, http://www.poforwarding.org [retrieved: March, 2015].

[17] Martin Casado and Teemu Koponen and Daekyeong Moon and Scott Shenker, "Rethinking Packet Forwarding Hardware," in *ACM SIGCOMM HotNets Workshop*, November 2008, pp. 1-6.

[18] Ran Giladi, "Network Processors: Architecture, Programming, and Implementation (Systems on Silicon)," *Morgan Kaufmann*, 2008.

[19] Open Compute Project, http://www.opencompute.org/ [retrieved: March, 2015] .

 ISBN: 978-1-61208-404-6