OPEN NETWORKING
FOUNDATION

OF-PI: A Protocol Independent Layer

Version 1.1
September 5, 2014

ONF TR-505

ONF Document Type: Technical Reference (TR); IP Assertion Type II
ONF Document Name: OFPI_Protocol_Independent OpenFlow_v.1.0_072014

Disclaimer

Revision History

0.4	6/16/14	Converted to Google Doc. Still initial drafting of content.
0.5	6/20/14	Major structure in place; changed to OFPI
0.51	6/23/14	Added “structure” figure; renamed “run-time API”.
0.6	7/1/14	Re-organized into proposal form, main flow: OF-PI language, SDN ecosystem, evolution from OF1.x
0.7	7/7/14	Removed repetition in first few sections. Propose to move Prior Work section to end.
0.8	7/10/14	Editor’s cleanup and resolution of recent comments.
0.9	7/15/14	Clarified Figure 4 and relation to OF1.x.
1.0	7/15/14	Draft for circulation.
1.1	9/5/14	Draft with revisions based on CAB, TAG, Chairs feedback.

1. Introduction

The promise of SDN is that a single control plane can directly control an entire network of switches. That is, instead of having network behavior “baked-into” network switches and other devices this behavior can be defined by software developed outside of (often proprietary) switch platforms. While network devices could be programmed in many ways, having a common, open, vendor-agnostic interface (like OpenFlow) enables a control plane to control network devices from different hardware and software vendors. However, today's OpenFlow is designed primarily with run-time control of commonly available switches in mind. This focus has enabled rapid deployment of OpenFlow in a variety of network applications; but it has limitations as well. In particular, the OpenFlow ecosystem currently does not address programming the datapath functionality in a switch.

We can divide SDN programming into two distinct stages: datapath configuration and run-time control (see Figure 1). The configuration stage defines the datapath functions provided by a switch and the run-time control stage uses these functions to control traffic flows through the switch.
Currently OpenFlow switches recognize a pre-determined set of header fields and process packets using a small set of predefined actions. Essentially the datapath configuration is pre-determined (assumed) by the protocol. As a consequence, expanding the capability of OpenFlow to control additional network technologies leads to continual modification of the OpenFlow protocol specification. As SDN becomes more widely adopted, there is pressure to evolve the OpenFlow protocol, to extend the protocols it supports, the packet headers it recognizes, and the way in which packets are processed. On its current trajectory, OpenFlow can be expected to become more and more complicated over time.

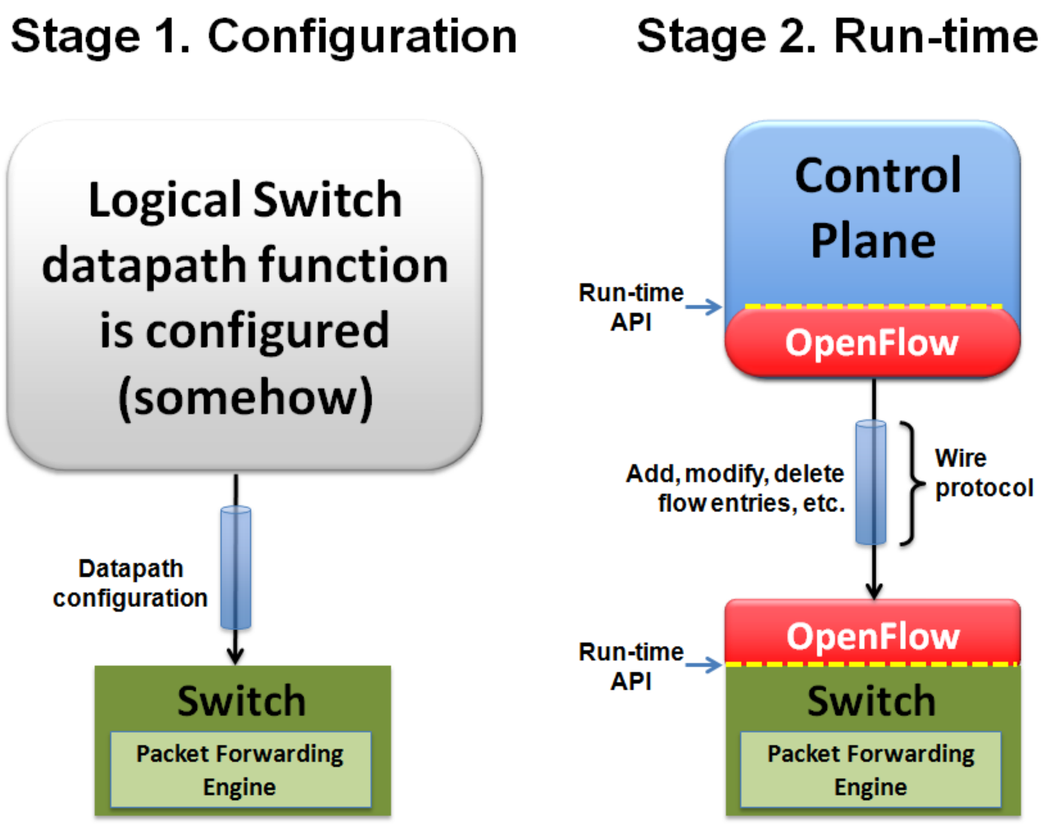

Figure 1: The OpenFlow protocols (to date) do not address datapath configuration.

While the current trajectory may be well-suited to fixed-function switch chips, which evolve slowly, we believe a different approach is needed for more flexible/programmable switches. With programmable switches the datapath configuration stage can be addressed directly as a programming problem. There is growing interest in more flexible and programmable forwarding planes and, in addition to software switches, the spectrum of flexible hardware implementations continues to become more colorful. For example, SwitchBlade [23] defined protocol-independent packet processing pipelines using FPGAs as a target. Intel's FlexPipe architecture has been flexible for some time, as described in [16]. The RMT paper [13] describes the architecture of a flexible multi-stage packet processor. Xpliant is believed to be working on a programmable network ASIC [14]. Corsa Technology has announced programmable network devices [15]. Xilinx has developed a programming language for defining how an FPGA-based switch processes packets [10]. And of course NPU-based forwarding equipment is widespread with examples from EZChip [17] and Netronome [18].

By definition, a flexible switch allows a developer (or control plane) to define how packets are to be processed "top down" rather than a switch ASIC defining it "bottom up" at the time the chip is designed. While both approaches have their merits, we believe there is sufficient interest in flexible switches to ask the question: *Can we define a common and vendor-agnostic way for developers to describe how packets should be processed by a flexible switch, in a manner that is consistent with OpenFlow*? This whitepaper outlines one possible approach. The scope of this white paper is restricted to addressing configuration of these flexible switches.

Our proposal builds primarily on two prior activities: Protocol Oblivious Forwarding (POF) [1] and Programming Protocol-Independent Packet Processors (P4) [2]. Both POF and P4 were designed with programmable switches in mind. POF proposes a protocol independent instruction set that allows a developer to express much more flexible packet processing than the current OpenFlow specifications. P4 provides a framework and higher-level language for programming a switch in terms of an instruction set such as the one proposed by POF.

The heart of our proposal is a new "Protocol Independent" layer, OF-PI, which would be capable of expressing both existing and new datapath protocols, as shown in Figure 2. OF-PI comprises a protocol-neutral instruction set, called POF-FIS, and the P4 language, for expressing how a forwarding plane should process packets. A P4 program describes the legal packet headers and how packets are to be processed by a switch.

P4 and POF-FIS could be used to express the functionality supported by existing OpenFlow specifications (e.g. OF1.0, 1.1, …) allowing them to be expressed as library modules on top of OF-PI. (Backward compatibility is not quite complete at this stage, because of some differences in the way actions are defined. The differences are discussed in Section 8). Existing protocols (e.g. IPv4, MPLS, GRE, etc.) could also be expressed as P4/POF modules, using P4 to define the packet formats and table structure and the protocol independent POF-FIS primitive instruction set to perform protocol-specific actions. These modules could be composed to create a datapath model for a switch capable of processing a given set of protocols. For example, the set of protocols included in each of the OpenFlow specifications could be described using a P4 program with POF-FIS primitive instructions.

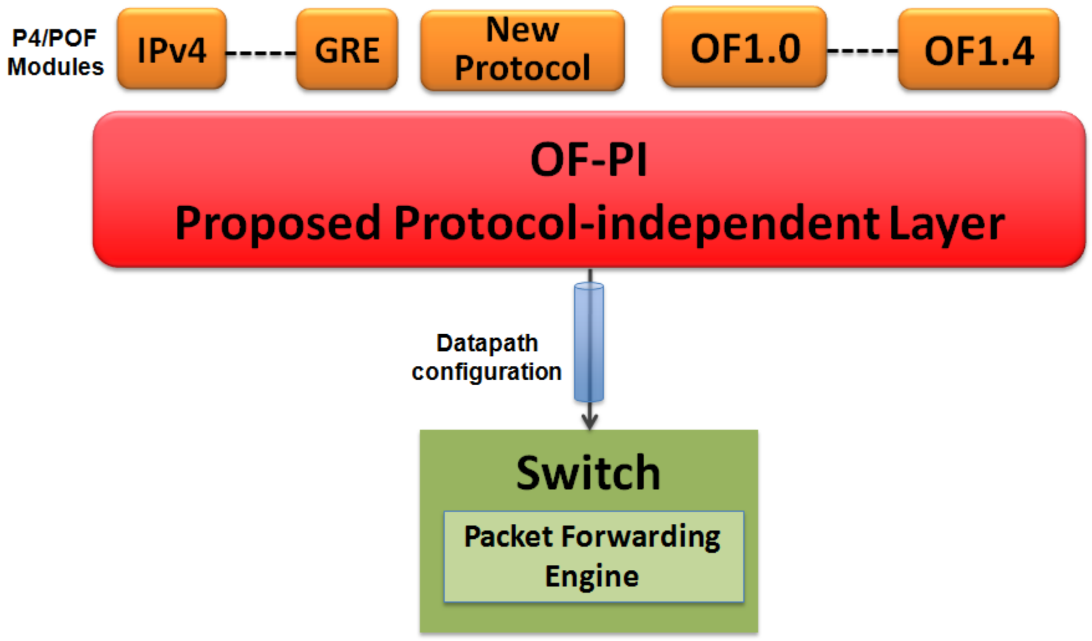

Figure 2: Proposed "Protocol Independent Layer", OF-PI.

The ONF Forwarding Abstractions Working Group (FAWG) [12] created Table Type Patterns (TTPs) so that a chip vendor or network architect can describe the profile of features and pipeline structure a switch supports, “bottom up”. Figure 3 compares how a TTP can represent the capabilities of a pre-defined datapath, while a P4/POF program defines the behavior of a switch datapath and, ideally, can be compiled into a loadable form to configure the datapath of a programmable switch, “top-down”. A TTP can be used to specify the OpenFlow controllable capabilities of a switch, describing the datapath to a controller. A P4 program might configure the switch to operate as forwarding device compatible with some OpenFlow version, or define a collection of existing and new protocols. Both languages represent datapath configuration, but approach it from a different perspective. At run-time a controller should not detect any difference between a fixed-function platform conforming to a specific TTP and a programmable platform configured by a P4 program to provide the same behavior. In Section 8 we describe the relationship between OF-PI and existing OpenFlow protocols in more detail.

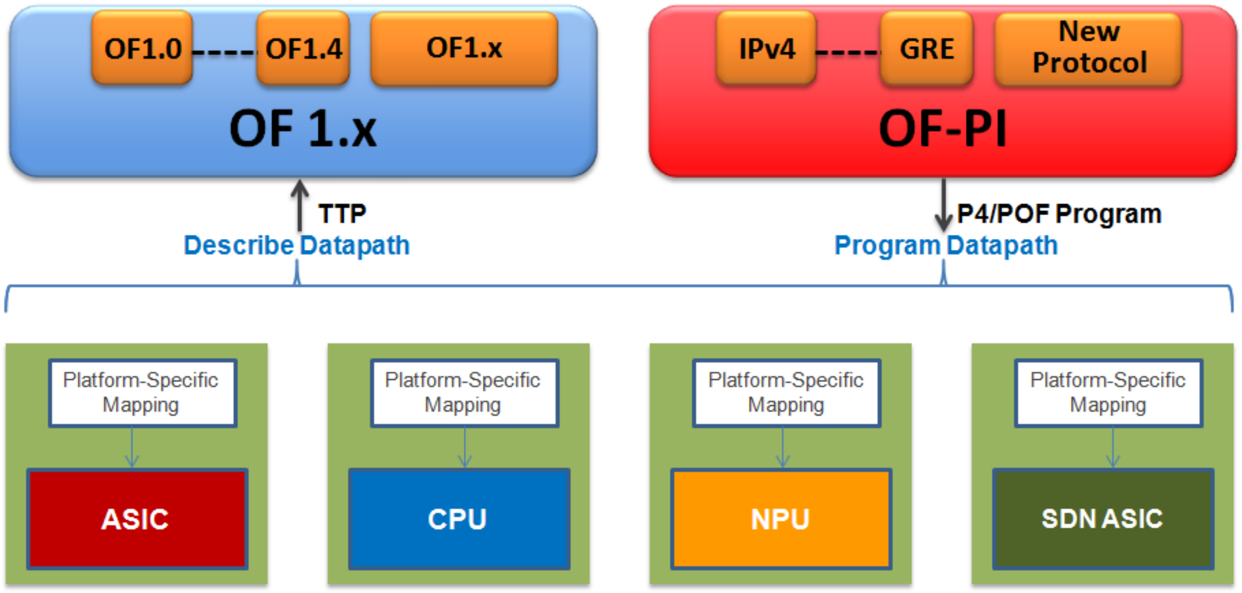

Figure 3: The relationship between TTPs and OF-PI.

The rest of this whitepaper presents a technical recommendation that ONF develops and eventually specifies OF-PI. We emphasize that our strawman proposal is incomplete, and we offer it as a contribution to the discussion about how to program more flexible switches.

2. The Goals for OF-PI

OF-PI is proposed with three goals in mind:

- **Reconfigurability**. The switch programmer should be able to re-configure the packet parsing and processing in the field on platforms that support flexible functionality.

- **Protocol independence**. OpenFlow should not be tied to specific packet formats. Instead, the switch programmer should be able to specify (i) a packet parser for extracting header fields with particular names and types and (ii) a collection of typed match+action tables that process these headers.
- **Target independence**. Just as a C programmer does not need to know the specifics of the underlying CPU, the switch programmer should not need to know the details of the underlying switch. Instead, a compiler should take the switch's capabilities into account when turning a target-independent description of how packets are to be processed into a target-dependent program (used to configure the switch).

We believe that future generations of switch programmers should be able to *tell the switch how to operate*, rather than be constrained by a fixed switch design.

3. Guiding Principles for OF-PI

The goals above led us to three guiding principles for OF-PI:

- **OF-PI should be protocol neutral** – Our proposal for OF-PI is based on POF-FIS which does not define any protocol-specific packet formats or actions. P4 provides a language for defining packet header formats, packet parsing rules, and datapath actions based on low-level instructions. The primitive instruction set allows programmers to program the datapath processing for both existing and new packet formats and behaviors.
- **OF-PI should help create a software development ecosystem** – Existing OpenFlow specifications reflect a "bottom-up" design process in which the capabilities of the forwarding plane are determined by fixed-function chips. Over time, we expect SDN to become more "top-down", with a programmer defining how the network is to process packets in a high level language. A compiler - for various switch platforms - will automatically generate a switch configuration. The same language may be used to describe pre-defined datapaths "bottom-up" for controlling newer fixed-function switches. By adopting common software practices, we believe OF-PI will lead to software reuse, tool suites, and the adoption of software engineering best practices.
- **OF-PI supports existing OpenFlow specifications** – Basing OF-PI on a more flexible programming model enables the definition of datapath models that are equivalent to the way OF 1.0 and OF 1.3 are used. The proposed more general approach may lead to some protocol or datapath modeling aspects differing from those specified in the existing OpenFlow protocols; however, as far as possible the next generation should be designed to be backward compatible to provide an effective evolution path for the existing OpenFlow software base.

Based on these guiding principles, we present a proposed solution in the following sections.

4. The Structure of OF-PI

We start by recognizing the two stages of programming a switch: configuration and run-time (see Figure 4). The configuration stage defines a datapath model including the format of packets to be handled, the actions that can be performed on these packets, the conditions under which these actions are invoked (match+action tables) and the necessary order of processing (match and data dependencies). The datapath configuration can be programmed, compiled and loaded into a programmable switch. With the datapath configuration in place, the run-time stage includes creating/modifying/deleting table entries through an API, passing packets between the switch and the controller, and other run-time actions.

At a high level, we propose that OF-PI includes the configuration stage (including the language to express how packets are processed, and the primitive actions performed on packets), and a run-time API (which might be autogenerated from the configuration). We don't propose a specific wire protocol (as part of OF-PI), because it will depend on how the switch is being used. For example, if a user programs the switch to replace an OF1.0 switch, then it may make sense to use the OF1.0 wire protocol. We leave it as an open question as to where the wire protocol is specified, but suggest that it does not belong in an OF-PI specification.

4.1. Configuration

The switch configuration is defined by a program written using the P4 language. The configuration programming stage enables the programmer to define the following:

- **Headers**: A header definition describes the sequence and structure of a series of fields. It includes information about field location and size, constraints on field values, determination of the presence or absence of optional fields, etc.
- **Parsers**: A parser definition determines the presence and order of headers within a packet.
- **Actions**: The construction of complex actions from simpler protocol-independent primitives. These complex actions are available within match+action tables.
- **Tables**: Match+action tables are the mechanism for packet processing. The program defines the fields on which a table may match and the actions it may execute. The program may also allow decorating a table with additional attributes like key size, capacity, search rate, etc. that provide compilers with useful hints for optimizing mapping to physical platforms.
- **Control Flow**: The control flow determines the order of matches and actions that are applied to a packet (i.e., the datapath pipeline structure) allowing for parallel execution where possible.

A program may be written from scratch (e.g., to describe new headers and new processing), or might be composed from one or more libraries (e.g., libraries describing existing protocols such as IPv4, VLAN, etc.). Ideally, P4/POF programs are target-independent and can be used to describe how packets are processed in a variety of different switches from different vendors. P4 programs are compiled by a P4 compiler to generate a target-specific switch configuration, which is then downloaded to the switch. The switch configuration can be downloaded in bulk or configured interactively using POF extensions to OpenFlow. Typically, the switch is configured at boot-time. But OF-PI should not require configuration to happen only once: a switch may support reconfiguration during operation, and we expect future switches will allow some reconfiguration without interrupting packet processing.

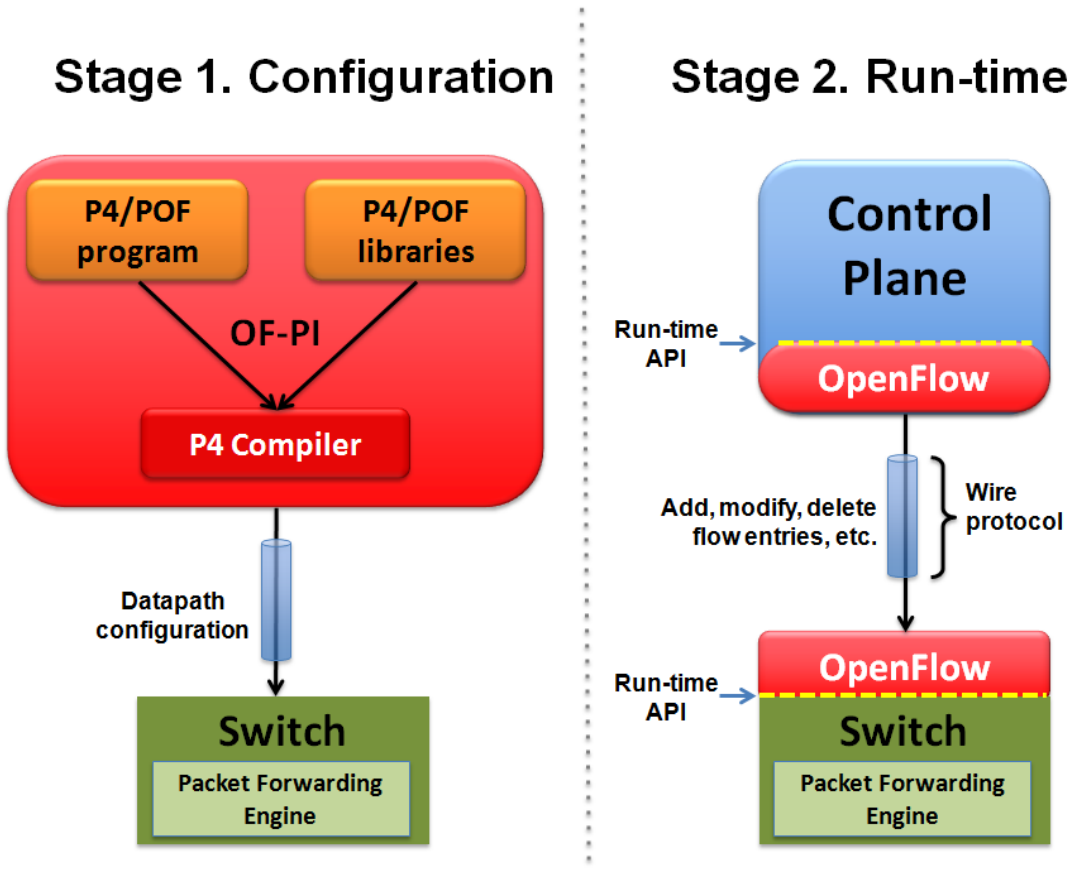

Figure 4: OF-PI has two stages, Configuration and Run-time.

4.2. Run-time

Once the switch is up and running, the control plane uses a Run-time API and associated wire protocol to interact with the switch. For example, the control plane might add and delete flow entries. Notice that in order to add and delete flow entries, the control plane and the switch need to agree upon the format of flow entries and tables, which are defined in the P4 program. Therefore, the Run-time API depends on the switch configuration. A P4 compiler therefore auto-generates the Run-time API for adding, modifying, and deleting flow entries for each format it recognizes. The Run-time API can be used in a controller to invoke OpenFlow commands and in a switch to implement those commands. Also notice that in Figure 4 we separate the “wire protocol” from the Run-time API. Currently the OpenFlow protocol specifies the set of commands available and the wire format for the control plane to send the commands to the switch. In OF-PI we propose to separate the marshaling of data (e.g., Restful JSON) from the Run-time API so that both can evolve separately. However, this does not preclude the ONF from specifying particular wire protocols and Run-time APIs. For example, the OF1.0 specification could be rewritten as a P4 library (to describe the recognized header fields and actions), a

Run-time API (the set of commands in the OF1.0 spec) and a wire protocol (the OF1.0 wire protocol between the control plane and switches).

5. A Language for Configuring Switches

To enable a switch datapath to be programmed, we propose that OF-PI relies on an abstract forwarding model and language designed specifically for this purpose. P4 and POF (and OpenFlow) employ an abstract forwarding model based on Match+Action Tables (MAT) that contain rules for processing packets received by the switch. Figure 5 depicts a simplified form of an OF-PI abstract forwarding model. (Note that this is one example and other, more or less detailed, abstract models should be explored to find a model well-suited for OF-PI.)

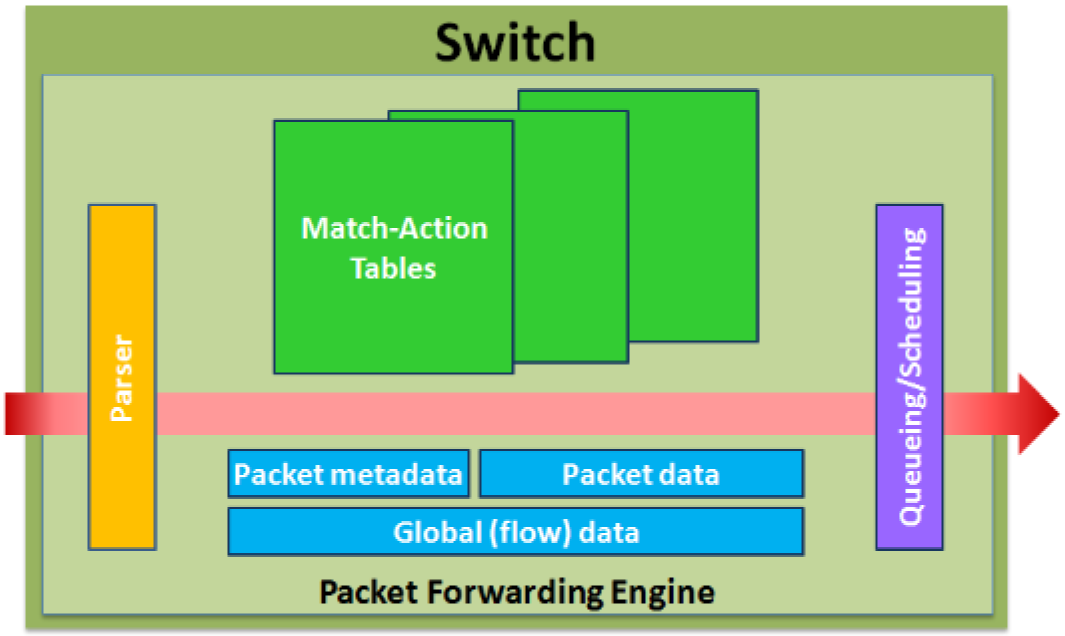

Figure 5: Abstract packet forwarding engine.

Packets received by the switch are first parsed to obtain the initial header contents (i.e., recognize header fields relevant to packet processing) and set initial packet metadata (e.g., ingress port number). Packet processing proceeds through a sequence of MATs, each of which may perform actions that can access packet data, packet metadata, and global data (e.g., flow state preserved by the switch) and may perform additional packet parsing. Finally, a packet is assigned to zero or more queues to be scheduled for transmission on egress ports. Figure 5 identifies some essential components of an abstract forwarding model; however, there are a variety of ways that these components can be realized in an implementation and the figure is not intended to imply any particular implementation structure.

OF-PI enables a programmer to program the packet forwarding engine configuration. P4 and POF provide a starting point for developing a complete OF-PI language. The following sections describe the essential features of the OF-PI language.

5.1. Header field definitions and packet parsing

OpenFlow relies on implicit header field definitions and parsing rules. That is, the protocol supports reference to header fields but does not provide a mechanism to formally describe packet header structure, or how the fields in the header are to be processed by the switch. To enable SDN to take full advantage of programmable platforms including the ability to define new packet formats, and new ways of processing fields, this implicit information must become explicit.

OF-PI provides a programmer with the capability to define header formats using the P4 language. This allows a programmer to define new headers and header fields and valid packet formats (header sequences) that are to be processed by a switch. The same parsing language can be used to describe the headers recognized by the OpenFlow specifications. Header definitions and packet formats can be organized into libraries to be included when creating a P4 program that uses the defined headers and packet formats. (The definition of P4 is a work in progress and will continue to evolve; it is used here to illustrate what a packet-processing language might look like.)

Making the header formats and parsing rules explicit allows for packets to be parsed immediately upon arrival, or incrementally as new information becomes available. Up-front parsing is most efficient when the set of allowed packet formats is known *a priori*. In some cases, however, a portion of the packet format is not known until a packet is parsed and inspected (e.g., an inner header type depends on the value of an outer header label). In these cases, the parsing function may be revisited. Both up-front and incremental parsing should be supported by OF-PI.

5.2. Protocol independent instruction set

Programming in OF-PI is based on a set of protocol-neutral primitives, which are low-level instructions that allow programming of a wide variety of datapath functions and do not assume any pre-configured protocol behavior in the switch. An example of this kind of primitive instruction set is the POF Flow Instruction Set (FIS) [1]. (POF FIS is a work in progress and is likely to evolve over the next few months. It is sufficiently stable at this point to illustrate what an OF-PI primitive instruction set might look like.)

The primitive instruction set should be minimal, including instructions such as:

- insert/remove header
- set/copy field
- add/sub/inc/dec field
- boolean operations and logical shifts on fields
- some notion of function calls (e.g., invoking h/w accelerators)
- execution controls (e.g., jump, conditional jump, goto table)

A “field” is a header field, per-packet metadata, or global data (e.g., for storing flow state information). Fields can be referenced using a low-level addressing scheme (e.g., offset from a defined location and length). A jump instruction (conditional or unconditional) allows simple logic to be programmed in action lists instead of requiring additional tables. Some more complex instructions are useful as well, for example to calculate a checksum or hash over selected packet data. The primitive instruction set provides a flexible foundation for OF-PI, and is augmented with higher-level constructs as described in the next two sections.

5.3. Protocol-specific functions

Once the header fields and packet formats have been defined, OF-PI enables a programmer to describe how packets are to be *processed* by the switch, in terms of the protocol independent instruction set.

OpenFlow specifies a set of protocol-specific actions well-suited to handling common datapath protocols (e.g., Ethernet, MPLS, IP). These actions include pushing and popping tag headers, reading and writing specific header fields, etc. While we propose removing the protocol-specific actions from the OpenFlow-Switch specification, the SDN ecosystem should continue to support actions defined at a level convenient for handling specific datapath protocols.

OF-PI provides the ability to define parameterized functions that perform commonly used actions for a given datapath protocol. These protocol-specific actions can be programmed using the protocol independent FIS (e.g., as POF instruction blocks or P4 action functions). It is also possible to define protocol-specific functions based on reference to a standard or other unambiguous behavioral specification. This latter approach is useful for fixed-function switch platforms that do not benefit from a function definition in FIS. Protocol-specific functions can be used to make match+action programming more convenient and to reduce the overall code space required in the switch. Allowing functions to have parameters increases the ability to reuse code by increasing the size of reusable code blocks. Protocol-specific function definitions can be organized into libraries that can be referenced (included in a datapath program) for switches that need to support the associated datapath protocol.

5.4. Autonomous functions

OpenFlow does not directly support functions that operate outside the match+action datapath. For example, while it is acknowledged that proactive monitoring can occur, OpenFlow does not provide for direct control of these functions. For example, ports have an operational state that can be derived from link monitoring, but how the monitoring is done is left unspecified in OpenFlow.

OF-PI should provide the ability to define autonomous datapath function objects with internal state. This enables delegation of a variety of control processes to the switch (e.g., MAC learning, ARP, link discovery, monitoring, protection switching, etc.) while maintaining the SDN principle of separation of control and datapath functions. On fully programmable platforms autonomous functions can be programmed and downloaded to the switch. On fixed-function platforms that have built-in support for such functions these functions can be controlled by an external controller even though it is not possible to (re)program them. This has the benefit of enabling efficient implementation of control functions that are not always practical to locate in a physically distant controller. Autonomous function objects can also be used to control stateful flow processing, e.g., stateful firewall functions. A proposal for “packet processors” [6] expresses a similar concept and may provide a starting point for development of this capability in OF-PI.

To support autonomous datapath objects a more complete event framework is needed, including support for timers. In the current OpenFlow-Switch protocol the main event is the receipt of a packet at a port. This event triggers the pipeline processing defined by flow tables. There is also implicit support for status events (e.g., the notion of port operational state changes) associated with Fast Failover groups. OF-PI should have an explicit and comprehensive event framework that can be used flexibly to initiate actions defined by the datapath configuration.

5.5. Match+Action Tables

As with OpenFlow, the application of actions to packets is governed in OF-PI using a series of MATs. These tables contain match conditions that recognize packet flows and apply appropriate actions to the associated packets based on entries programmed by the SDN controller at run-time. The MAT model is preserved in OF-PI, albeit with extensions allowing the controller to delegate repetitive, time-constrained functions to the switch (e.g., using the autonomous functions mentioned in the previous section). The MAT model provides a convenient mechanism for run-time programming of flows in a well-defined datapath model. The ability to create, modify, and delete table entries that implement common flow patterns makes run-time network flow configuration both efficient and predictable.

5.6. Control Flow Representation

The representation of the flow of a packet through the tables is key to representing the behavior of the datapath. Network programmers using this framework should be able to use traditional structured programming techniques to represent this control flow. In this approach, each table is an instruction and control structures such as conditionals or loops could be used to describe the order in which the instructions are applied. Underlying the flow, however, the table ordering can be viewed as a graph.

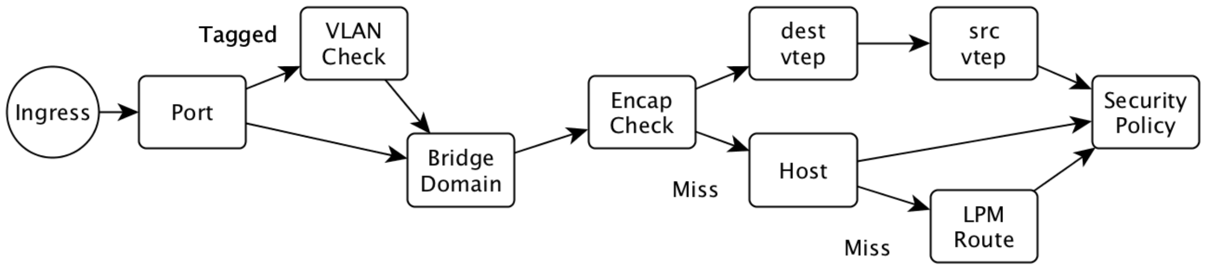

Figure 6: An Example Table Dependency Graph.

The diagram above shows an example table dependency graph for a simple ingress pipeline. Each node represents a table. The graph, in conjunction with the table declarations, captures the data dependencies intended by the programmer. For example, the *Bridge Domain* table may set a metadata field that is read by the *Encap Check* table. Without this ordering information, the compiler may not be able to infer that the operations specified by the *Bridge Domain* table must be completed before the *Encap Table* reads its match data.

Each edge in the graph represents a control decision and may be annotated by the conditions under which the edge is traversed. This representation is amenable to analysis by traditional compiler tools and mechanisms and provides an opportunity to optimize the ordering of data modification dependencies. Most importantly, table dependency graphs may be targeted to different implementations. Thus, we propose that the Table Dependency Graph be used as an intermediate representation for the control flow with programmers being allowed to express their intent by writing an imperative flow program.

6. An SDN Ecosystem

Switches can be programmed in many ways and having a common, open, vendor-agnostic interface (like OpenFlow) enables a single logically centralized control plane to control switches from many different hardware and software vendors. However, one protocol is not sufficient to create an SDN marketplace in which many vendors (hardware and software) offer interoperable products to network operators. The growth of an SDN marketplace will depend on a number of elements that can be collectively called the "SDN ecosystem." A software ecosystem supports both (1) software-specific activities—e.g., coding, translation, linking/loading, initialization, operation, and upgrade—and (2) broader development activities—e.g., specification, simulation, development, conformance testing, migration, and debugging.

The SDN ecosystem supported by OF-PI includes a language for configuring the switch datapath (e.g., P4/POF-FIS), libraries of reusable datapath configuration modules for specific datapath protocols, a run-time API for controlling a configured switch datapath, and a controller-to-switch protocol allowing separation of the control plane and the datapath hardware (e.g., OpenFlow).

6.1. SDN programming

The context in which SDN control is exercised influences the ecosystem requirements. Many approaches can be taken to programming an SDN switch. These are derived from choices made by the programmer, driven in part by the capabilities of the target platform and the software ecosystem. Each approach follows a similar sequence for defining the software elements (identified in section 4) that control network behavior. There is a natural sequence of definition:

1. header formats (and other fields),
2. protocol-specific actions,
3. table types (match+action combinations),
4. control flow (datapath pipeline structure), and finally

5. table entries.

Different programming approaches vary the time and method of defining these elements and may implement some element definitions in hardware instead of software. Our goal with OF-PI is to define an SDN ecosystem that supports many approaches on as broad a variety of switch platforms as is practical.

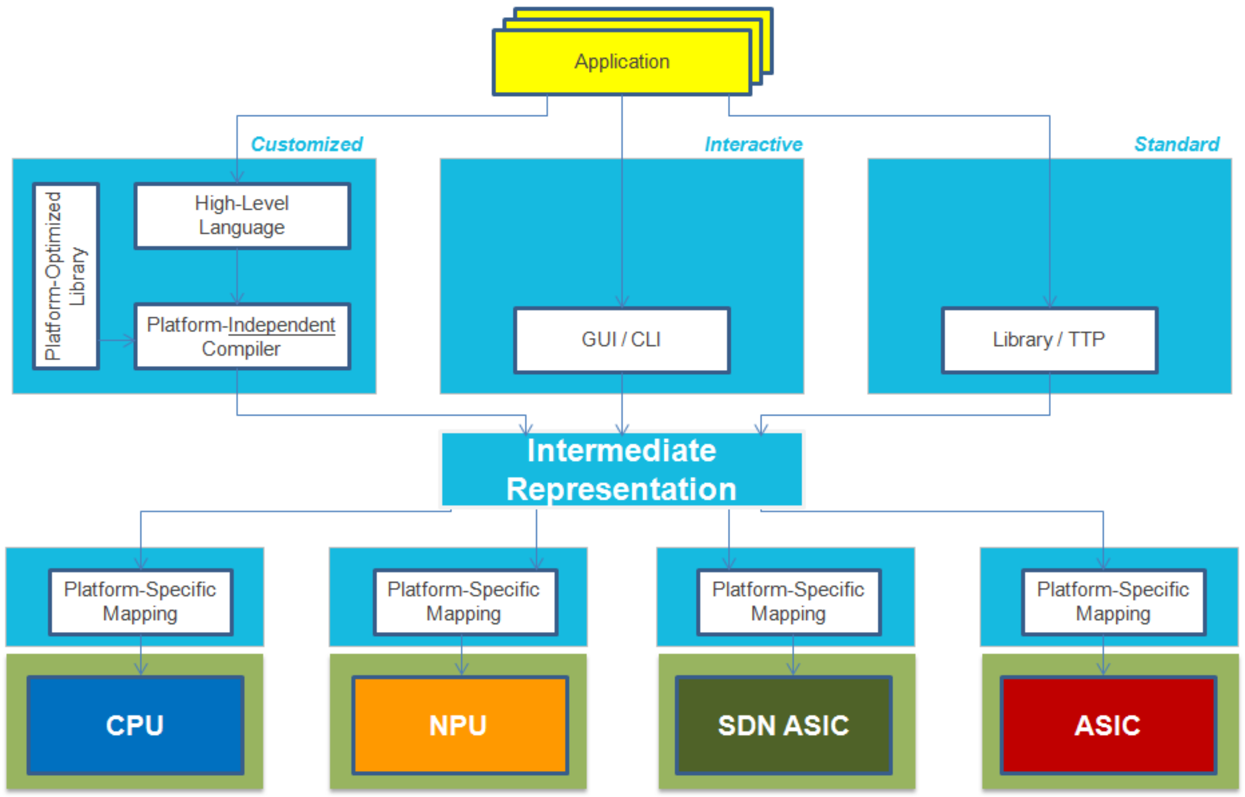
Application
Customized
Interactive
Standard
Platform-Optimized Library
High-Level Language
Platform-Independent Compiler
GUI / CLI
Library / TTP
Intermediate Representation
Platform-Specific Mapping
Platform-Specific Mapping
Platform-Specific Mapping
Platform-Specific Mapping
CPU
NPU
SDN ASIC
ASIC

Figure 7: SDN programming approaches.

Figure 7 shows three approaches to SDN network configuration and run-time control:

- **Standard** – controlling a network that employs a standard, pre-configured datapath (e.g., a fixed-function Ethernet/MPLS/IP switch);
- **Interactive** – network research and network debugging/repair/extension allowing dynamic datapath configuration;
- **Customized** – SDN development for production networks based on formal specification and testing of new datapath models.

If the datapath behaviors required of a switch are defined by a published standard, one can specify a datapath model based on that standard that allows a controller to create flows that conform to the standard. The header formats, parsing rules, actions, and table pattern are stable and supported by standard libraries. This stability allows implementations to commit protocol-specific behaviors to hardware to improve performance. Therefore the configuration phase can include development of

ASICs that are fixed well before run-time. Each switch platform can incorporate software (e.g., a platform-specific SDK) to simplify translation from a standard OpenFlow datapath model to its own ASIC configuration.

Approach	Configuration Phase	Run-time Phase
Standard	Design ASIC and write SDK	OpenFlow-Switch
Interactive	OpenFlow-Switch	OpenFlow-Switch
Customized	Write P4 program, compile, load	OpenFlow-Switch

Table 1: Example SDN programming approaches

The interactive approach is most applicable to research networks where flexible configuration is a key requirement and to SDNs that allow flexible operational policies. In this approach the OpenFlow-Switch protocol can support the entire range of switch programming from configuration through run-time (the POF prototype [9] provides an example). In the interactive approach all aspects of configuration (defining header formats, actions, tables, etc.) as well as run-time commands are provided via one interface and the network operator can adjust the datapath model on-the-fly. This generally requires a fully programmable switch platform like a CPU, NPU, or one of the more recent SDN-focused switch designs.

Production networks using flexible SDN technology may take a more deliberate approach to network design and operation. In a customized approach the network architecture and associated switch datapath models can be developed and tested before deployment in a production network. SDN datapath models customized to a particular network application can be written and then compiled into loadable configurations for the SDN network switches. The configuration stage might include downloading switch configurations or simply negotiating the datapath model when the OpenFlow-Switch control channel is initialized. The run-time stage is essentially the same as in other approaches; however, making a change to the datapath configuration in a production network may require controlled upgrade procedures. For example, a change to the datapath configuration may involve thoroughly testing the new configuration before installing it in the network and performing a graceful upgrade of the datapath configuration in each switch to avoid interrupting network traffic.

The variety of approaches to SDN development creates a need for a variety of artifacts, tools, and processes in an SDN ecosystem. Some of these are described in the following sections.

6.2. Configuration stage

An important element in the SDN ecosystem is support for specifying and configuring a switch datapath. With OpenFlow Table Type Patterns (TTPs) can be used to represent an OpenFlow controllable datapath model. A TTP is useful for specifying the controllable aspects of a fixed-function switch or an agreed switch behavior in a well-defined network architecture. However, in their current form TTPs are not sufficient for fully automated configuration of a flexible switch platform. The

OF-PI language capabilities described in section 5 allow for configuration stage programming of flexible switch platforms.

POF defines extensions to the OpenFlow wire protocol that allow a programmer (or controller) to reference arbitrary data fields (both packet and metadata), define protocol-specific functions, and create match+action tables and table entries. This provides an entirely interactive capability for configuration of a flexible switch platform that can be very effective for network research, development, and debugging.

Using the OF-PI datapath configuration language enables the creation of compilers that transform a datapath program into a form that can be installed on a particular switch platform. These compilers can be constructed from a platform-independent stage that parses the OF-PI language and transforms the program into an intermediate representation (IR) based on the FIS, and a platform-dependent stage that transforms the IR into a platform-specific configuration module. The platform-specific stage may take advantage of technology developed to map header specifications to flexible parse engines and map MATs to platform pipeline stages and memory blocks with varying capabilities (e.g., DRAM, SRAM, TCAM, etc.).

For example, the OF-PI ecosystem should include tools to compile a P4/POF datapath specification into a loadable configuration that can be installed in a programmable switch. Reusable libraries containing datapath protocol parse and MAT modules should be available for direct reuse or customization by the programmer. The compilation process can also produce a customized run-time API and supporting code for generating and parsing wire protocol messages between the controller and switch.

6.3. Run-time stage

The run-time stage can use the OpenFlow-Switch protocol. The basics are already in place – messages to create, modify, and delete flow table entries. However, the OF-PI ecosystem can be enhanced by including some extensions to simplify repetitive tasks (which are quite common in the run-time stage).

The datapath configuration programming model allows for the definition of higher level actions with parameters (e.g., see 5.3). However, each table entry (match+action combination) is currently created as though it is a unique entity. A datapath model (e.g., a P4/POF program or a TTP) often restricts table entries to a few specific types. The table entry type names in a datapath model can be used (with appropriate parameterization for the variables in the table entry definition) to simplify the creation of table entries. If a datapath model is in use, table entries can be created using just the table entry type name and variable parameters instead of sending a fully elaborated table entry. This is essentially syntactic shorthand and neither increases nor decreases the flexibility of the OpenFlow-Switch control protocol.

A second form of syntactic shorthand is already partially included in OpenFlow-Switch 1.4 – bundles. A bundle is a set of OpenFlow commands collected into a set that is treated as atomic. That is, either all the commands in the set are successfully executed or none are executed. However, the current specification indicated that a bundle is deleted once it is executed. It is common in some networks to have patterns of flow entries that create a useful network service. For example, a bi-directional connection requires two flow entries, one for each direction. Extending the bundle concept into a more general control macro that is parameterized and persistent would provide useful syntactic shorthand for creating frequently used patterns.

By adding these elements of syntactic shorthand OF-PI can minimize redundant or unnecessary control traffic without impacting overall SDN flexibility.

6.4. Libraries and Protocols

In OF-PI, protocols are written as P4 programs; for example, one program for IPv4, and others for GRE, VLANs and so on. One intriguing possibility is that future switch developers might combine the protocols they need from several libraries (public or proprietary). This could make possible code reuse and rapid prototyping of new network systems.

For this to be possible, the libraries need to be composable, in the sense that each protocol library is self-contained and may be assembled with others without modification. This brings with it many of the benefits associated with modern software development; for example, independently developed, tested and re-used libraries; self-contained and therefore more resilient modules; and possibly the deployment of new functionality into an operational system. would make possible the development of functional modules with a focus on the interfaces between them, the basis of modern software development. This, in turn, supports the definition of more resilient systems and the integration of new functionality even while operational.

6.5. Information model

Finally, a formal information model may be a useful addition to the SDN ecosystem, for example to allow marshalling OpenFlow-Switch messages in a variety of useful wire formats (e.g., XML and JSON in addition to the currently defined C-structure format). This could apply to both configuration information and run-time information. While a formal information model is not strictly required the benefits to the SDN ecosystem can be substantial by enabling development of more powerful tools and thus aiding the development of SDN products.

7. Relationship to Prior Work

The concepts incorporated in this proposal draw from a number of sources. Our primary sources are the work on POF [1] and P4 [2]. However, the ideas can be traced back to earlier proposals for more flexible programming from Google [4][5] , and Xilinx [10][11], and others (e.g. [3][6][7][8]), as well as academic papers investigating languages for such platforms [20][21][22].

The ONF Forwarding Abstractions Working Group (FAWG) [12] created TTPs so that a chip vendor or network architect can describe the profile of features and pipeline structure a switch supports, “bottom up”.

Figure 3 showed how a Table Type Pattern (TTP) can represent the capabilities of a fixed function datapath and a P4 program can define the capabilities of a programmable datapath. A TTP is used to specify the flexibility of a given datapath design in terms of OpenFlow controls (flow tables, flow entry types, group table entry types, etc.). Thus a TTP provides a developer with a specification of datapath behavior that they can use to guide coding of controller or switch agent software. On the other hand, a P4 program defines the behavior of a switch datapath and, ideally, can be compiled into a loadable form to configure the datapath of a programmable switch.

The OpenFlow-Switch protocol can be used to control a switch datapath whether the switch platform is fixed-function or programmable. In fact, at run-time the controller may not detect any difference between a fixed function platform conforming to a specific TTP and a programmable platform configured by a P4 program to provide the same behavior.

The OF-PI proposal does not incorporate all of the ideas proposed above, but we do think the direction set here is flexible enough to accommodate most, if not all, of the capabilities suggested. The proposal presented here is intended as a starting point for discussion and development of OF-PI and we welcome suggestions for improving the OF-PI concept and its realization.

8. Backward compatibility and evolution

Protocol independence is intended to enhance (and not replace) the existing OpenFlow specification. To this end, maintaining backwards compatibility with the OpenFlow protocol, where possible, is an explicit goal. OF-PI does not dictate changes to the OpenFlow-Switch protocol. Thus, controllers may be updated to provide datapath configuration while continuing to support existing OpenFlow Logical Switches; similarly, new programmable switches may continue to interact with existing OpenFlow controllers, though additional functional modules in the control plane may be required to configure a datapath if it is capable of being programmed.

The abstract model of forwarding defined in OF-PI is consistent with the match+action approach of OpenFlow. The concepts of port, table, match, action, and control channel (that is, an interface allowing the manipulation of the tables that govern the forwarding behavior of the pipeline dynamically) remain unchanged.

OF-PI provides a means of evolving from the current fixed specification of fields and operations. It supports identifying and defining the headers and fields relevant to a particular Logical Switch and the protocols it needs to support. This is done in conjunction with a specification of the actions needed to effect the operations necessary for those protocols.

OF-PI takes a different approach from OpenFlow in how actions are defined. In our proposal, actions are defined in the configuration stage. OpenFlow has no configuration stage, so all actions are specified when an entry is added at run-time. It would be feasible to extend the proposal to allow actions to be specified at run-time or for the choice of action to be based on conditional tests at run-time.

9. Next Steps and Recommendations

This whitepaper came about because several of the authors were independently trying to answer the same question: *Can we define a common and vendor-agnostic way for developers to describe how packets should be processed by a flexible switch, in a manner that is consistent with OpenFlow?* We were aware of several approaches, and came to learn that we were thinking about it independently in different groups (POF and P4). We came together to see if we could identify common goals, principles, and ideas; and we were pleasantly surprised to discover that the approaches we had been pursuing independently were very complementary. POF provides guidance on what the underlying primitive instruction set should be, and P4 provides guidance on how packet-processing programs can be written and compiled, using the POF instruction set. The complementary nature of the two approaches, and our desire to answer the question above, led us to come together and propose the OF-PI approach.

While a lot of effort has been put into developing POF and P4 over the last two years, neither is fully-baked or complete. Many questions remain unanswered; for example, how a programmer might define how packets are buffered and scheduled, and how statistics are maintained. We definitely do not think this is the last word on OF-PI; it is just the beginning of what we hope will be a healthy technical discussion, and we hope our proposal can provide a concrete strawman to launch the discussion within ONF. We are committed to finding an answer to the original question, and look forward to working with the ONF community to evolve our approach to best answer the question above.

To this end, we are circulating this whitepaper among ONF members in late July 2014, to solicit feedback. We would love to hear from members interested in starting an ONF "OF-PI" Working Group dedicated to answering our original question.

10. References

[1] H. Song, "Protocol Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane," in *SIGCOMM HotSDN Workshop*, Aug. 2013.

[2] P. Bosshart, et. al., "P4: Programming Protocol-Independent Packet Processors," in *ACM SIGCOMM Computer Communication Review*, July 2014.
http://arxiv.org/pdf/1312.1719v3.pdf

[3] D. Cohn, "OpenFlow Futures – Entry Points and Legacy Modeling,"
https://rs.opennetworking.org/wiki/download/attachments/1769725/OF-Futures22212.pdf?api=v2.

[4] N. Yadav, et. al., "OpenFlow Future Requirements (Forwarding Model and Language),"
https://docs.google.com/document/d/161Z44oYzYwaQqn2iCragglEM0jSG27yuzEZZ6izKeXA/edit?pli=1.

[5] N. Yadav, D. Cohn, "OpenFlow Primitive Set," https://docs.google.com/document/d/1v_l0MCPBZ1RBWYnIeN-2v9Pix0HUCtXPCJKOhwKaRRA/edit?pli=1.

[6] Z. Kis, "PACKET PROCESSORS," https://rs.opennetworking.org/bugs/secure/attachment/18101/packet-processors-v2.pdf.

[7] J. Tönsing, "Scenarios Informing Model Expressiveness Part 1: Layering + Tunnels," https://rs.opennetworking.org/wiki/download/attachments/2228667/of-futures-2012-05-30.pdf?version=1&modificationDate=1338398204000&api=v2.

[8] M. Yu, A. Wundsam, M. Raju, "NOSIX: A Lightweight Portability Layer for the SDN OS," http://www.google.com/url?q=http%3A%2F%2Fwww.sigcomm.org%2Fccr%2Fpapers%2F2014%2FApril%2F0000000.0000003&sa=D&sntz=1&usg=AFQjCNH16PRrGWxgsYfq7ZszyYh2AIv-XQ

[9] Protocol Oblivious Forwarding, http://www.poforwarding.org

[10] N Possley, "Software Defined Specification Environment for Networking" http://www.xilinx.com/applications/wired-communications/sdnet.html

[11] M. Attig and G. Brebner, "400 Gb/s Programmable Packet Parser on a Single FPGA", http://www.xilinx.com/innovation/research-labs/conferences/ANCS_final.pdf

[12] The Forwarding Abstractions Working Group Charter. http://goo.gl/TtLtw0, April 2013.

[13] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN," in ACM SIGCOMM, 2013. http://yuba.stanford.edu/~nickm/papers/RMT-SIGCOMM.pdf

[14] R. Merrit, "Will ASICs be replaced in comms gear?", http://www.eetimes.com/document.asp?doc_id=1263181, April 2013.

[15] C. Matsumoto, "Corsa Builds a Purely OpenFlow Data Plane", http://www.sdncentral.com/news/corsa-builds-purely-openflow-data-plane/2014/02/, February 2014.

[16] "Intel Ethernet Switch Silicon FM6000," http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch- fm6000-sdn-paper.pdf.

[17] "EZChip 240-Gigabit Network Processor for Carrier Ethernet Applications." http://www.ezchip.com/p_np5.htm.

[18] "Netronome Launches Data Plane Hardware and Software for SDN and NFV Designs" http://www.netronome.com/march-4-2014-netronome-launches-data-plane-hardware-and-software-for-sdn-and-nfv-designs/

[19] H. Song, J. Gong, H. Chen, "Coherent SDN Forwarding Plane Programming", in ONS 2014

[20] Nate Foster, Michael J. Freedman, Arjun Guha, Rob Harrison, Naga Praveen Katta, Christopher Monsanto, Joshua Reich, Mark Reitblatt, Jennifer Rexford, Cole Schlesinger, Alec Story, and David Walker, "Languages for Software-Defined Networks," in *IEEE Communications Magazine*, volume 51, number 2, February 2013.

[21] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker, "Modular SDN Programming with Pyretic," in *USENIX ;login:*, October 2013.

[22] Cole Schlesinger, Michael Greenberg, and David Walker, "Concurrent NetCore: From Policies to Pipelines", in *International Conference on Functional Programming (ICFP),* September 2014.

[23] Anwer, Motiwala,Tariq, and Feamster, "SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware", in *SIGCOMM 2010*, August 30-September 3, 2010.

11. Contributors

The following people have contributed to this white paper: Ben Mack-Crane (editor), Ben Pfaff, Curt Beckmann, Dan Talayco, Haoyu Song, Jennifer Rexford, Justin Dustzadeh, Nick McKeown, Yatish Kumar.