DC.p4: Programming the Forwarding Plane of a Data-Center Switch

Anirudh Sivaraman*, Changhoon Kim†, Ramkumar Krishnamoorthy†, Advait Dixit†, Mihai Budiu†
*Massachusetts Institute of Technology, †Barefoot Networks
anirudh@csail.mit.edu,{chang, kram, adixit, mbudiu}@barefootnetworks.com

ABSTRACT

The P4 programming language [29, 16] has been recently proposed as a high-level language to program the forwarding plane of programmable packet processors, spanning the spectrum from software switches through FPGAs, NPUs and reconfigurable hardware switches. This paper presents a case study of using P4 to express the forwarding plane behavior of a datacenter switch, comparable in functionality to single-chip shared-memory switches found in many datacenters today.

This case study allows us to understand how specific P4 constructs were useful in modeling specific datacenter switch features. We also outline additional language constructs that needed to be added to P4 to support certain features of a datacenter switch. We discuss several lessons that we learned in the process and distill these into a proposal for how P4 could evolve in the future.

CCS Concepts

•**Networks** → **Programmable networks;**

Keywords

Programmable forwarding planes; language design; datacenter switch;

1. INTRODUCTION

The Internet has long prided itself on an architecture that stresses smart endpoints and a dumb network [32, 45]. Over time though, with evolving application requirements, the network has become smarter with network switches implementing functions such as access control, tunneling, and overlay formats. A typical switch today implements functionality that in aggregate covers over 7000 RFCs. This number is growing by the day as new protocols [12, 24, 5] are standardized.

This large feature set is typically implemented by a switching chip [6, 11] with dedicated hardware to support these features. These chips support some flexibility by having a few configurable knobs, but are inherently not future-proof: a new protocol format usually necessitates an expensive redesign of the chip that can easily take a few years given the long time scales in hardware design. This is perhaps best illustrated by the long lag time between the standardization and availability of VXLAN [24], a recently proposed encapsulation format.

More importantly, the switch's functionalities and capabilities are dictated by the switch vendor and not the network operator, who is likely in the best position to know exactly how the switch is going to be used.

Over the last few years, several trends suggest a move towards a "top-down" view dictated by the network operator, as opposed to the "bottom-up" view mandated by the switch vendor. First, software-defined networking (SDN) [43] standardizes a common interface to the switch control plane and allows operators [38] to write their own control applications using that interface. Second, emerging switch architectures such as the RMT architecture [30] and commercial switching chips such as Intel's FlexPipe [7] and Cavium's Xpliant [25] embrace field programmability as an explicit goal. Third, languages such as P4 [29, 17] and POF [47] provide language support to express this architectural flexibility. Fourth, recent research on compilers [40] for these languages seeks to bridge the gap between the language and the hardware.

With the availability of language, architecture, and compiler support for programmable forwarding planes, it is natural to ask what useful applications can be built with a programmable forwarding plane? To answer this question, we set out to express the forwarding behavior of a datacenter switch using P4. The resulting P4 program exceeded 2500 lines of code and is likely the largest software artifact expressed in P4 yet. This allowed us to understand how well P4 serves its purpose as a packet-processing language.

We begin this paper by reviewing (§2) the P4 language proposal and abstract switch model [29, 16] and describe our development environment (§3) to compile and execute P4 programs. We then focus on one specific P4 program (§4), DC.p4 that expresses the forwarding behavior typical of a datacenter switch today (Table 1). This exercise resulted in several new P4 language features that were added to the original proposal for P4 [29] and are now part of the P4 language specification [16]. Based on our experience with DC.p4 (§5), we propose a pathway for evolving P4 (§6).

The P4 program used throughout this paper is open source and is available, along with directions to compile and execute it, at http://git.io/sosr15-p4. The P4 compiler front end

ACM 978-1-4503-2836-4/14/08 ...$15.00.
http://dx.doi.org/10.1145/2619239.2626324

and the software switch that runs the compiled program are also open source and available at http://p4.org/code.html We hope these software artifacts encourage wider adoption of P4 and lead to the formation of an open-source community driving P4 development.

2. P4 PRIMER

This section briefly reviews P4's abstract switch model and language constructs to provide context for the rest of the paper. For a more detailed look at P4 and its syntax, we encourage the reader to consult either the paper describing P4 [29] or the P4 specification [16]

As a design goal, P4 tries to be portable in two ways. First, P4 programs are expected to be target-independent, allowing a programmer to specify packet processing uniformly across different implementation platforms: software switches [41], FPGA-based switches [3], NPUs [9], and reconfigurable hardware [30, 7] switches. Second, the P4 language itself is protocol-independent in that it doesn't natively model any specific protocols. Instead, it provides abstractions that allow a programmer to express both existing and future protocol formats in a common syntax.

Abstract switch model: P4 assumes an abstract switch model where a switch has a programmable parser to parse packets from incoming bits on a wire. Packets then enter an ingress pipeline consisting of a sequence of match-action tables. Each match-action table can modify the packets that it sees before sending it to the next one. At the end of the ingress pipeline, packets from multiple input ports are switched to their output ports by a buffering subsystem. Packets then enter an egress pipeline that also consists of a sequence of match-action tables for further packet processing. Finally, a deparser serializes the packets onto the wire.

Headers: The programmer specifies a set of header definitions that the switch is expected to process. A header definition specifies the fields belonging to that header, their sizes, and the order of the fields within that header. For instance, the Ethernet header specifies the 6-byte source and destination MAC addresses and the 2-byte EtherType.

Parsers: The parser abstraction specifies how a switch processes incoming packets to populate headers within these packets. P4 models the parser as a state machine that processes headers sequentially within a packet and expects the programmer to provide the order of headers within the packet. For instance, the programmer would specify that the IPv4 header follows the Ethernet header.

Actions: P4's actions generalize OpenFlow's [43] fixed action set (forward, drop, mark, etc) to a user-specifiable set of actions. These flexible action descriptions are composed from a small set of primitive actions. For instance, a compound action on a packet, "set_next_hop", could be composed out of two primitive actions: one that sets the next hop's MAC address and another that decrements the IP TTL field.

Tables: Tables generalize OpenFlow's match-action abstraction to match on any subset of the header fields parsed by the parser. In contrast to OpenFlow, the fields on which matches are possible are not fixed and limited to existing protocol formats; rather, they can be any field within the headers specified by the programmer. The P4 table abstraction specifies the fields to match on (such as source or destination IP or MAC address), the match policy (such as exact match, ternary matches with wildcards, or longest-prefix match), the size of the table to assist the compiler back-end [40], and the allowed set of actions for this table. As an example, longest-prefix IP routing would match on the IP destination field, with the action setting the next-hop MAC address and decrementing the TTL.

Control flow: Lastly, P4 provides an abstraction for control flow i.e. the order in which tables must be processed on an incoming packet. For instance, the programmer could specify that IP Access Control be applied before IP routing. Control flow can also be conditional where a packet is sent to the next table only if a specific boolean predicate on the packet fields is true.

3. A P4 DEVELOPMENT ENVIRONMENT

We briefly describe our development environment for writing, compiling, and debugging P4 programs as context for the rest of the paper. A subsequent paper will discuss the development environment in greater detail; here, we only summarize the main features to provide enough context for the discussion to follow.

The programmer writes P4 programs using the syntax described in the P4 specification [16]. The programmer can also use C macros such as #define and #include within the P4 program. These macros are first expanded by the C preprocessor and the output of the preprocessor is passed to a P4 compiler. The P4 program is then compiled by the P4 compiler front end that checks for syntax errors and undefined references. The compiler also analyzes the P4 program to determine table dependencies mandated by the control flow. Table dependencies are of two forms: data dependencies, where a second table matches on a field that is set by an action in the first, and control dependencies, where the second table's execution is predicated on a field set by the first table.

In this paper, the compiler target for our P4 programs is a *software* switch, to ease the task of prototyping. In the future, we expect P4 will be used to program the forwarding plane for a variety of targets such as the RMT architecture [30] and Intel's FlexPipe [7]. For a software switch target, the table-dependency graph is used to generate C code that is functionally equivalent to the P4 program. This C code is compiled into a static library, which is then linked with a software switch core that can read and write packets from a virtual Ethernet interface. The linking produces a software switch that executes the P4 program. The software switch limits the rate of packet dequeues from the shared buffer to the egress pipeline to emulate a specific link rate.

The compiler also auto-generates a run-time API for the forwarding plane that can be used by a controller to populate table entries at run time. Presently, this API uses the Thrift RPC [2] framework for Remote Procedure Calls, although it could be extended to generate an API resembling either OpenFlow [43] or the Open Compute Project's Switch Abstraction Interface [13, 20]. A block diagram of the software switch and its relation to the controller is shown in Figure 1.

The software switch can be used as a drop-in replacement for Open vSwitch [15] within Mininet [42]. This allows us to create topologies of such P4-programmed switches for larger tests. Within this topology, the switches forward traffic between end hosts just like Open vSwitch, allowing us to test the P4 programs with both closed-loop traffic such as congestion-controlled TCP and open-loop traffic such as test

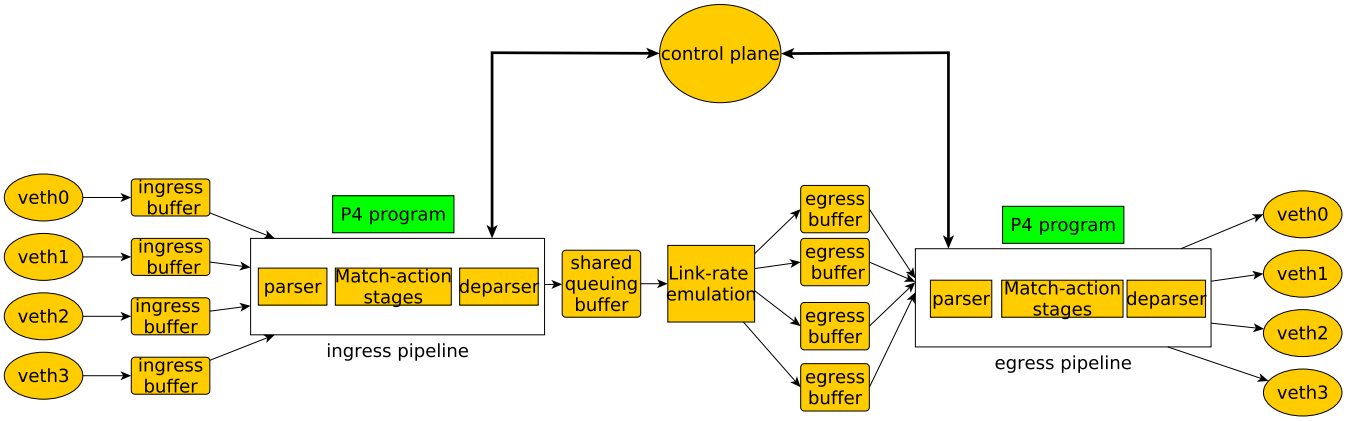

Figure 1: Block diagram of 4-port software switch that executes P4 programs. The switch communicates with other switches or end-hosts within Mininet through the veth interfaces, which are Linux Virtual Ethernet Interfaces [23]. For scalability, packet processing is handled by a thread pool with each thread in the pool responsible for a subset of ports.

packets generated by OFTest [14]. We have used both approaches to test the P4 program presented in this paper.

4. A DATACENTER SWITCH IN P4

We now look at a complete P4 program, DC.p4, which captures the forwarding behavior of a datacenter switching chip (Table 1 captures the feature set covered by this program). In total, the program contains over 2500 lines of P4 code, 25 ingress tables, 12 egress tables, and a table dependency graph (i.e. the graph induced by control and data dependencies between tables) with a diameter of 10. This program has substantial complexity, measured both in the number of lines of code and the complexity of its dependency graph, allowing us to evaluate P4 in a realistic setting.

Code for DC.p4 is available at http://git.io/sosr15-p4. It consists of the main file DC.p4 that specifies the program's control flow. Within the "includes" folder, headers.p4 contains header definitions and parser.p4 specifies the parser state machine. The "table_definitions" folder contains table definitions for the ingress and egress pipelines. Throughout this section, we use P4 files from this code base to explain specific language features.

We first look at features from Table 1 that could be captured by P4's original constructs (§4.1) and then move on to features that required new language constructs (§4.2).

4.1 Features captured by the original language

Many features listed in Table 1 can be expressed naturally using the P4 constructs described in the original P4 proposal [29]. For instance, IP forwarding (ip_fib.p4) based on either exact, ternary, or longest prefix matches is expressed through a table that uses one of the three match types when matching on the destination address. Similarly, packet validation, such as forbidding Martian Packets [10] and packets with options, in validate_packet.p4 is also expressed easily as an exact or ternary match on the appropriate field.

4.2 Language Changes

Header stacks: Variable-length headers, where there could be zero or more instances of the same header type, are common in real network protocols. MPLS and VLAN tags are a case in point. The original proposal for P4 [29] doesn't model this explicitly. We now model these using an array of header instances, where each instance is of the same header type. An example is provided in port_vlan_mapping.p4

Counters: So far, our discussion of P4 hasn't dealt with state in the forwarding plane that must be persisted across packets. In practice, however, persistent state in the form of counters is useful in tracking how many times a particular routing table entry has been encountered, or how many packets were dropped by an ACL rule. To capture these requirements, we introduced the notion of counters. mac_ip_acl.p4 illustrates an example using counters to track the number of matches for every entry in the ip_acl table.

Field lists: Field lists are an abstraction to represent a group of fields that are combined together to produce a single scalar value. One use case of a field list is in computing hashes based on the 5-tuple, used for randomized load balancing in Equal Cost Multi Path routing (ECMP) [1],

This example is shown in ecmp_group.p4. Here the field list consists of the flow's 5-tuple and the accumulate operation on the field list (represented by the field_list_calculation keyword) specifies a CRC-16 hash across all fields.

Match-select-action tables: Match-action tables are now ubiquitous in software-defined networking. The match-action abstraction assumes a one-to-one mapping from a match to an action.

Some forwarding-plane features don't fit within this abstraction. The ECMP example above (ecmp_group.p4), for instance, needs to assign a matched flow to one among a set of available ports for effective load balancing. The action in this case is the output port selection, and isn't static; it is decided dynamically based on a random hash of the 5-tuple.

To model this, we introduced the notion of action profiles: a table of actions that a match-action table can index to dynamically select an action at run time. The process of action selection is modeled by the action selector construct. In the ECMP case, this is ecmp_hash, a random hash of the 5-tuple to determine the runtime action.

Action profiles can also be used without a selector to refer to an action from a common set of actions. This is useful when multiple table entries share both actions and action

Pipeline	Functionality	Language constructs	Source code
Ingress	Virtual LANs (VLANs)	Action profiles, header lists	port_vlan_mapping.p4
Ingress	Spanning Tree Protocol	Exact match	spanning_tree.p4
Ingress	Common logic to handle routing for NVGRE [12], VXLAN [24], and ERSPAN [5] tunnels	Exact match	outer_rmac.p4 ipv4_dest_vtep.p4 ipv4_src_vtep.p4 tunnel.p4
Ingress	Packet validation	Exact match	validate_packet.p4
Ingress	ECMP	Action profiles, Action selectors, Field lists	ecmp_group.p4
Ingress	IP forwarding	Longest-prefix match	ip_fib.p4
Ingress	Link Aggregation (LAG)	Action profiles, Action selectors, Field lists	lag_group.p4
Ingress	MAC and IP Access Control Lists	Counters	mac_ip_acl.p4
Ingress	Packet Mirroring	Clone packet	mirror_acl.p4
Ingress	MAC learning	Digest Generation	learn_notify.p4
Egress	Tunnel decapsulation for NVGRE, VXLAN, ERSPAN	Add headers	tunnel_decap.p4
Egress	Tunnel encapsulation for NVGRE, VXLAN, ERSPAN	Remove headers modify_field_with_hash	tunnel_rewrite.p4 tunnel_src_rewrite.p4 tunnel_dest_rewrite.p4
Egress	VLAN tag add/removal	Header lists	egress_vlan_xlate.p4
Egress	MTU Check	Ternary match on mtu_check_fail field	egress_system_acl.p4
Egress	Process packets to/from switch CPU	Add/remove header	cpu_rewrite.p4

Table 1: Feature list of DC.p4 along with language constructs and corresponding source files

parameters. vlan_port_mapping.p4 provides an example of this latter use case.

New actions: P4's premise is that a small set of primitive actions can be composed together into larger compound actions. This is largely true: of the 90 compound actions in DC.p4, 83 are composed entirely from primitive actions such as setting field values, addition or subtraction on fields, copying into and out of fields, and adding or removing fields.

That said, we had to add a few new primitive actions to support some of the features in Table 1. The new primitive actions and their use cases are listed below:

1. **Packet cloning** is used to clone packets. mirror_acl.p4 shows an example. The mirror_acl table matches on a set of packet fields to determine which packets to clone. The action for this table clones the packet using the primitive clone_ingress_pkt_to_egress and sets a session id to denote the mirroring session (the encapsulation format and destination port of cloned packets).
2. **Packet dropping** drops packets and is used for access control (system_acl.p4, egress_system_acl.p4, and egress_block.p4)
3. **Digest generation** creates a digest containing specific fields in the packet. This primitive produces a concise representation of the packet without modifying the packet in flight and is useful for learning: where the control plane is made aware of the existence of a new flow. Specific examples of learning are the MAC learning functionality in learn_notify.p4, used to learn new MAC addresses and OpenFlow's Packet-In [19, 21] mechanism that notifies the SDN controller whenever there is a miss in the match-action table.
4. The **modify_field_with_hash** primitive sets a specific packet field to a hash value generated by an algorithm like CRC-16. It is useful when tunneling and ECMP are combined. For example, let's assume a router performs ECMP by computing a hash on the packet's 5-tuple. When tunneling and ECMP are combined, the 5-tuple will be identical for all flows with the same tunnel end points, causing uneven load balancing. However, there is sufficient diversity within the 5-tuples of the *encapsulated* packets, which can still be exploited for load balancing. A CRC-16 hash on the encapsulated 5-tuple can be used to set the packet's outer UDP source port, in the process creating sufficient diversity for load balancing. tunnel_rewrite.p4 illustrates this.

4.3 Features that are not yet modeled in P4

P4 was not originally designed to express how packets are scheduled (e.g. strict priorities, WFQ [34], etc). This is because—unlike packet forwarding, multipath routing, and access control—packet scheduling requires looking at multiple packets concurrently to determine which packet to schedule next. Therefore, today, P4 treats packet scheduling as a black-box. Given the many new ideas that use packet scheduling to improve application performance in datacenters [26, 27, 37] it is natural to ask how P4 can be extended.

5. LESSONS LEARNED

We set out to evaluate P4 through one specific case study: the forwarding plane of a datacenter switch. While a datacenter switch is just one example of a forwarding plane, the exercise has taught us several lessons, described below.

5.1 What P4 gets right

P4 has the right level of abstraction for many packet-processing tasks. Proposals such as Protocol-Oblivious Forwarding [47, 18] manipulate the entire packet header using lengths and offsets, requiring tedious bit manipulation from the programmer. In contrast, P4 has language support for specifying both packet headers and packet parsing. This allows the programmer to think in terms of header fields rather than bits. It also allows the compiler to autogenerate the packet parser, eliminating several sources of error resulting from manual parsing [28].

By making tables first-class citizens, P4 benefits from the success of OpenFlow's match-action table abstraction. Match-action tables are a natural construct for many network programmers because forwarding behavior (routing, ACLs, tunnels) has been structured as a table lookup for decades now.

Lastly, P4 has a simple cost model: the per-packet latency, memory footprint, and maximum throughput of a packet-processing program can all be determined at compile time. The latency cost of a control-flow dependency is equal to the processing delay of a single stage in a switch pipeline. The memory footprint of an additional table is given by the table size in the P4 program. If the P4 program fits in a given switch backend, its maximum throughput is limited only by the maximum processing rate of the underlying hardware, regardless of the program itself. Several language features contribute to this simple cost model: unlike packet processing based on imperative languages such as C, P4 forbids loops with unknown iteration counts and doesn't support dynamic memory allocation—both of which lead to performance variance at run time.[1]

5.2 Avenues for improvement

P4's current support for modularity and information hiding is limited to C's #include statements that can be embedded within P4 code to move unrelated parts of the program to separate files. This is a rather weak form of modularity: a table within the P4 program can manipulate any header field in its actions, leading to undesirable coupling between tables. P4 also doesn't make explicit the flow of information from one table to another: the programmer needs to look at both tables to determine what fields are written and read by each table.

P4 assumes parallel semantics for the execution of primitive actions within compound actions. Being closer to the way hardware works, this makes the compiler's job easier. In the process, it shifts the burden to the programmer, who has to reason about parallel-execution semantics. This is especially important considering that a large body of packet-processing software, such as the Linux qdisc subsystem, includes imperative code that is written assuming sequential semantics. If P4 will have to successfully span the spectrum from reconfigurable hardware to software switches, parallel execution semantics might be overly constraining.

While the language has well-defined types for packet headers and rigidly limits the permitted modifications on these headers, it is under-specified in parts. For instance, what is the behavior when an IP TTL field is decremented by 1 when it is already 0? Does it wrap around, become negative, or is it undefined? C has been plagued by such cases of undefined behavior in the past [48], and we would be remiss to repeat the same mistakes.

Expressing a datacenter switch in P4 has taught us that we need new primitives such as digest generation and cloning. However, adding primitives to a language to capture new requirements is something to be wary of. It risks bloating the language core with a large set of keywords that are opaque to—and, hence cannot be optimized by—the compiler.

6. LOOKING FORWARD

The lessons learned from using P4 to describe a datacenter switch suggest a pathway to evolve P4 in the future. We outline a few suggestions below.

6.1 Architecture-language separation

One important observation is that P4 could be much more than just a switch datapath description language. For instance, P4 could conceivably be used to describe the forwarding plane of a firewall or a network interface card (NIC). The P4 specification mandates that P4 be used to program switches implementing a fixed abstract forwarding model (Figure 1 in the P4 specification [16]). In practice, however, different packet-processing devices such as firewalls and NICs have very different packet-processing architectures.

To broaden the scope of P4 and accommodate such scenarios, we propose separating out the abstract switch model from the language definition. In this proposal, the switch model would be replaced by a model of the target architecture, without being restricted to switches alone. The target architecture model would be supplied by the target vendor and would not be part of the language definition. The target architecture is analogous to the concept of an instruction set architecture used by general-purpose CPUs today.

The target architecture model would identify the P4 programmable blocks in a target, along with non-programmable (fixed-function) blocks by defining the interfaces between the programmable and non-programmable blocks. Portability would then guarantee that a P4 program is portable across all targets that implement the same target architecture model.

Examples of fixed-function blocks would include packet scheduling, active queue management, and checksums. These fixed-function blocks would interact with other programmable blocks, such as match-action tables through well-defined interfaces specified by the target architecture model. Some new language features, such as counters and field lists, which had to be added to P4, (§4.2) could potentially be modeled as fixed-function blocks that implement an increment operation and a checksum respectively. These blocks would specify their complete behavior and input/output interfaces with other fixed-function and programmable blocks.

This would allow the core language to remain small by having fewer keywords. As a result, the language and target architecture models could evolve independently. This would permit organic growth of a variety of target architectures and their associated libraries containing fixed-function blocks. Instead of repeatedly extending the language to support new features, most additions would be made to libraries—providing a more sustainable alternative. Initially,

[1]In practice, the cost model also depends on the target: a software switch might emulate a ternary-match table using a trie instead of a hardware TCAM, causing lookup latencies to increase with table size. However, even for such targets, P4's restrictive constructs simplify the cost model relative to a Turing-complete language like C.

these blocks could be part of vendor-specific libraries, such as the Intel IPP [8]. If these blocks become widely supported, they could migrate into a standard library.

6.2 More complete language semantics

Certain aspects of P4 are incompletely specified. Examples include the overflow semantics of integers, exception handling, casting between different data types, and the initial values of table entries and packet attributes.

We propose that P4 standardize these behaviors. Practically, though, there is a tension between standardization and target independence. For instance, the overflow behavior might be different depending on the switch architecture. Mandating a specific overflow behavior could result in programs that work on certain architectures, but don't work on others.

6.3 Better support for modularity

Currently, P4 supports embedding #include statements that are processed by the C preprocessor, allowing a programmer to separate out distinct pieces of functionality. We propose that P4 be extended by borrowing modularity constructs such as lexical scoping and local variables from main-stream languages. One way to achieve this would be to structure all packet-processing functionality (parsing, deparsing, compound actions, match-action processing) as functions. The body of each function would only be allowed to access either its arguments or local variables, and nothing else.

7. RELATED WORK

7.1 Programmable forwarding planes

There has been recent interest in programmable forwarding planes for switches. The RMT architecture [30] is one example, and commercial offerings such as Intel's FlexPipe [7], Cavium's Xpliant [25], and Cisco's Doppler [4] feature similar programmable pipelines while also providing high performance. Academic work in this area includes the Tiny Packet Program interface [39] for low-latency network monitoring and control, PLUG [33], which provides a programmable set of lookup modules, and a proposal to add an FPGA for programmable queue management and scheduling [46]. Another approach to forwarding-plane programmability uses general-purpose processors [44, 35, 41]. Regardless of the underlying substrate (reconfigurable hardware, FPGA, or CPU), a universal high-level language such as P4 would make it easier to program different targets, without worrying about target-specific details.

7.2 Packet-processing languages

Protocol-Oblivious Forwarding (POF) [47, 18] is an ongoing project that shares many of the same goals as P4. POF treats packet headers as scratchpads that are accessed using {offset, length} tuples, resulting in a low-level programming model resembling assembly language. While this considerably simplifies the compiler, it shifts the burden of packet parsing to the programmer. P4 provides a high-level programming model that represents both packet headers and parsing in the language and generates the packet parser automatically.

packetC [36] is a domain-specific language for packet processing. packetC is more expressive than P4 and allows access to packet payloads, while P4 only permits header inspection and modification. Further, packetC also allows stateful processing by providing synchronization constructs for globally shared memory. P4's abstract switch model assumes shared state (such as counters) is local to a table and accessible only to the current packet. These differences stem from a difference in focus: P4 targets reconfigurable hardware platforms with limited flexibility and aggregate switching performance that exceeds 1 Terabit per second, while packetC focuses on NPUs, FPGAs, and software switches with much more flexibility and lower performance.

PX [31] is a packet-processing language that targets FPGA platforms such as the Xilinx Virtex-7 [22]. PX converts a high-level declarative specification of packet parsing and editing to a Register-Transfer Level (RTL) description of the target substrate in Verilog or VHDL. This RTL description is then further compiled to an FPGA bitstream by a standard FPGA tool chain. P4's abstractions for packet parsing and processing share some similarity with those of PX. However, by emitting RTL as its output, PX restricts itself to FPGA platforms, while P4 can span a variety of targets.

8. CONCLUSIONS

This paper presents a case study of using P4 to express the forwarding plane of a datacenter switch. Our findings suggest that the language can capture a good amount of datacenter switching functionality. Beyond this particular case study, expressing forwarding behavior in a high-level language has an important practical benefit: the functionality of the switch can be dictated by the network operator and not the switch vendor. Operator requirements change frequently and allowing the operator to change switching functionality improves responsiveness. Further, operators are no longer tied to a specific set of protocol formats that are supported by the vendor.

Architecturally, the approach of programming switches in a high-level language reduces switches to their core functionality of bit-stream processing. How this bit stream is turned into headers, then parsed into packets, and then processed against state resident on the switch is completely up to the switch operator. If this approach bore fruit, the switch's functionality would only be limited by the programmer's imagination, much like programming on CPUs today. We wouldn't have to restrict ourselves to a "dumb network" [45] and the network could benefit from innovation just like end hosts have in the past.

That said, much work remains before programmable forwarding planes become commonplace. This study explores one particular application: the datacenter switch. Whether P4 can express the forwarding plane of other switching equipment such as network interface cards, wireless access points, cellular base stations, and core routers is a question we leave for future work. We hope to have convinced readers that the approach of a programmable forwarding plane is a fruitful one. We encourage readers to extend our P4 program (available at http://git.io/sosr15-p4), write P4 programs for new use cases, and propose language improvements.

Acknowledgments

We thank Antonin Bas for developing the software switch, Leo Alterman for work on the P4 compiler front end, and John Cruz for debugging DC.p4

9. REFERENCES

[1] Analysis of an Equal-Cost Multi-Path Algorithm. https://tools.ietf.org/html/rfc2992.

[2] Apache Thrift - Home. https://thrift.apache.org/.

[3] The Arista 7124 FX as a High Performance Trade Execution Platform. http://www.argondesign.com/media/uploads/files/P8006-R-001d_The_Arista_FX_Switch_as_an_Execution_Platform.pdf.

[4] Cisco highlights next big switch. http://www.biztechafrica.com/article/cisco-announces-next-big-switch/5448/#.VP4mCYWltVZ.

[5] Configuring ERSPAN. http://www.cisco.com/c/en/us/td/docs/ios-xml/ios/lanswitch/configuration/xe-3s/lanswitch-xe-3s-book/lnsw-conf-erspan.html.

[6] High Capacity StrataXGS®Trident II Ethernet Switch Series. http://www.broadcom.com/products/Switching/Data-Center/BCM56850-Series.

[7] Intel FlexPipe. http://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

[8] Intel Integrated Performance Primitives (Intel IPP) | Intel Developer Zone. https://software.intel.com/en-us/intel-ipp.

[9] IXP4XX Product Line of Network Processors. http://www.intel.com/content/www/us/en/intelligent-systems/previous-generation/intel-ixp4xx-intel-network-processor-product-line.html.

[10] Martian Address Filtering. http://tools.ietf.org/html/rfc1812#section-5.3.7.

[11] Mellanox Products: SwitchX-2 Ethernet Optimized for SDN. http://www.mellanox.com/page/products_dyn?product_family=146&mtag=switchx_2_en.

[12] Network Virtualization using Generic Routing Encapsulation. https://msdn.microsoft.com/en-us/library/windows/hardware/dn144775%28v=vs.85%29.aspx.

[13] Networking/SpecsAndDesigns. http://www.opencompute.org/wiki/Networking/SpecsAndDesigns#Switch_Abstraction_Interface.

[14] Oftest. http://www.openflowhub.org/display/OFTest/OFTest+--+Validating+OpenFlow+Swtiches.

[15] Open vSwitch. http://openvswitch.org/.

[16] P4 Specification. http://p4.org/spec/p4-latest.pdf.

[17] P4.org. http://p4.org/.

[18] POForwarding. http://www.poforwarding.org/.

[19] Sdn / OpenFlow / Message Layer / Packetin / Flowgrammable. http://flowgrammable.org/sdn/openflow/message-layer/packetin/.

[20] Switch Abstraction Interface Specification v0.9. http://files.opencompute.org/oc/public.php?service=files&t=24b68e105629caf910d9b3f2834d7e6a&download.

[21] Understanding Openflow: Packet-In is a Page Fault. http://www.projectfloodlight.org/blog/2012/02/27/packet-in-is-a-page-fault/.

[22] Virtex-7 FPGA Family. http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html.

[23] Virtual Ethernet Interfaces. http://openvz.org/Virtual_Ethernet_device.

[24] Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. https://tools.ietf.org/html/rfc7348.

[25] XPliant^{TM}Ethernet Switch Product Family. http://www.cavium.com/XPliant-Ethernet-Switch-Product-Family.html.

[26] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pFabric: Minimal Near-optimal Datacenter Transport. In *SIGCOMM*, 2013.

[27] W. Bai, K. Chen, H. Wang, L. Chen, D. Han, and C. Tian. Information-Agnostic Flow Scheduling for Commodity Data Centers. In *NSDI*, 2015.

[28] J. Bangert and N. Zeldovich. Nail: A Practical Tool for Parsing and Generating Data Formats. In *OSDI*, 2014.

[29] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-independent Packet Processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[30] 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 *SIGCOMM*, 2013.

[31] G. Brebner and W. Jiang. High-Speed Packet Processing using Reconfigurable Computing. *Micro, IEEE*, 34(1):8–18, Jan 2014.

[32] D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *SIGCOMM*, 1988.

[33] L. De Carli, Y. Pan, A. Kumar, C. Estan, and K. Sankaralingam. Plug: Flexible Lookup Modules for Rapid Deployment of New Protocols in High-speed Routers. In *SIGCOMM*, 2009.

[34] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. In *SIGCOMM*, 1989.

[35] M. Dobrescu and K. Argyraki. Software Dataplane Verification. In *NSDI*, 2014.

[36] R. Duncan and P. Jungck. packetC Language for High Performance Packet Processing. In *11th IEEE International Conference on High Performance Computing and Communications*, 2009.

[37] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues Don't Matter When You Can JUMP Them! In *NSDI*, 2015.

[38] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a Globally-deployed Software Defined Wan. In *SIGCOMM*, 2013.

[39] V. Jeyakumar, M. Alizadeh, Y. Geng, C. Kim, and D. Mazières. Millions of Little Minions: Using Packets for Low Latency Network Programming and Visibility. In *SIGCOMM*, 2014.

[40] L. Jose, L. Yan, G. Varghese, and N. McKeown. Compiling Packet Programs to Reconfigurable Switches. In *NSDI*, 2015.

[41] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F.

Kaashoek. The Click Modular Router. *ACM Trans. Comput. Syst.*, 18(3):263–297, Aug. 2000.

[42] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, 2010.

[43] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, Mar. 2008.

[44] J. Perry, A. Ousterhout, H. Balakrishnan, D. Shah, and H. Fugal. Fastpass: A Centralized "Zero-queue" Datacenter Network. In *SIGCOMM*, 2014.

[45] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end Arguments in System Design. *ACM Trans. Comput. Syst.*, 2(4):277–288, Nov. 1984.

[46] A. Sivaraman, K. Winstein, S. Subramanian, and H. Balakrishnan. No Silver Bullet: Extending SDN to the Data Plane. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, 2013.

[47] H. Song. Protocol-oblivious Forwarding: Unleash the Power of SDN Through a Future-proof Forwarding Plane. In *HotSDN*, 2013.

[48] X. Wang, N. Zeldovich, M. F. Kaashoek, and A. Solar-Lezama. Towards Optimization-safe Systems: Analyzing the Impact of Undefined Behavior. In *SOSP*, 2013.