The Case for a Flexible Low-Level Backend for Software Data Planes

Sean Choi
Stanford University

Xiang Long
Cornell University

Muhammad Shahbaz
Princeton University

Skip Booth
Cisco Inc.

Andy Keep
Cisco Inc.

John Marshall
Cisco Inc.

Changhoon Kim
Barefoot Networks Inc.

ABSTRACT

Recent efforts to simplify network data plane programming focus on providing simple, high-level domain-specific languages (DSLs). In the case of software switches, data plane programs are written in these DSLs and then compiled to run on CPU-based architecture. However, the simplicity of these DSLs, along with the lack of low-level interfaces exposed by the software switch, restrict compilers from generating optimal data plane programs for CPU-based architecture.

In this paper, we argue that increased exposure of low-level interfaces to a software switch would enable more effective data plane programs. To demonstrate this, we present *Programmable Vector Packet Processor* (PVPP), which adds programmability to the Vector Packet Processing (VPP) framework. VPP provides fine-grain access to various low-level features of a CPU-architecture and offers better performance compared to other software switches, such as Open vSwitch (OVS), that operate at a higher level of abstraction. However, there is a cost to programming directly using VPP's low-level features. The programmer must have specialized knowledge about the architecture in order to produce an efficient implementation, resulting in difficulties when optimizing the program. PVPP attempts to alleviate this cost by allowing the compilation of a program written in P4 to VPP's internal node-graph representation. Our preliminary results show that PVPP improves performance of data plane programs by around 30% compared to naïve VPP implementations.

APNET '17, August 3–4, 2017, Hong Kong, China

ACM ISBN 978-1-4503-5244-4/17/08…$15.00
https://doi.org/10.1145/3106989.3107000

CCS CONCEPTS

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

KEYWORDS

Programmable Data Plane, Software Switch, P4, Vector Packet Processing (VPP), FD.io, PVPP

ACM Reference format:
Sean Choi, Xiang Long, Muhammad Shahbaz, Skip Booth, Andy Keep, John Marshall, and Changhoon Kim. 2017. The Case for a Flexible Low-Level Backend for Software Data Planes. In *Proceedings of APNET '17, Hong Kong, China, August 3–4, 2017,* 7 pages.
https://doi.org/10.1145/3106989.3107000

1 INTRODUCTION

There is an ongoing interest in network community for incorporating more programmability into the data plane. One particular area of interest is to improve methods for programming software switches, such as Open vSwitch (OVS) [8], that are widely used in the industry. The main use case of a software switch is in hypervisors, where network traffic is routed to and from virtual machines that the hypervisors manage. To program these software switches, protocol developers write network programs in domain-specific languages (DSLs), such as P4 [6], specifically designed to easily express packet processing logic. A target-specific compiler generates the final instructions executed by the software switch using the input program. PISCES [10], a P4 programmable version of OVS, is a good example of a switch that adopts this model of incorporating programmability.

However, we argue that if a compiler has fine-grain control over program instructions in the target software switch than what is available today, performance benefits can be gained for network programs. While existing software switches are designed and hand-tuned to support a particular set of features, they expose only a limited set of interfaces (*e.g.*,

OpenFlow [5]) for external entities (*e.g.*, controllers) to modify or fine-tune their functionality. This corresponds to the fact that, given their nature, DSLs are highly expressible in specifying packet forwarding logic but expressibility is limited for fine-tuning the target-specific parameters of the architecture. Thus, it is very difficult for compilers to specify low-level behaviors, such as defining memory and CPU cache access patterns or instruction parallelization behaviors in the software switch.

The goal of our paper is to make the case that improvements in performance, flexibility, and feature support are attainable if the target switch exposes more low-level interfaces. We offer the following contributions towards substantiating this claim:

- Review of the Vector Packet Processor (VPP) [11], a software switch that expose low-level interfaces. We explain how VPP is able to obtain better performance against competing switches, such as OVS, with the access to the low-level architecture. (Section 2).
- The design and implementation of Programmable Vector Packet Processor (PVPP), a P4 programmable software switch built using VPP framework. (Section 3).
- Compiler optimizations of PVPP by exploiting the low-level interfaces that VPP exposes. (Section 4).
- Evaluation demonstrating the increased performance of PVPP programs as more optimizations are made. We also present preliminary results comparing the performance of PVPP against the existing packet-forwarding features in VPP [11] and PISCES [10]. (Section 5).

2 VPP: A FLEXIBLE SOFTWARE SWITCH TARGET

We begin by giving an overview of the Vector Packet Processing (VPP) framework, the software switch that PVPP is built upon. The VPP platform [11] is an extensible packet processing framework that provides a fully-featured, highly-optimized forwarding engine designed to run on general purpose, commodity CPUs. The VPP platform runs completely in user-space by leveraging DPDK device drivers [2].

In VPP's vector processing model, the forwarding path is decomposed into a collection of processing nodes that are organized as a directed graph. Each node has the responsibility of processing an entire vector of packets, making local modifications and deciding the next node for every packet in the vector. Vector of packets are formed at the input nodes of the graph by processing as many available packets from the RX driver as possible. The main reason for having each node work on a vector of packets at a time is to optimize i-cache and d-cache locality across the entire vector, minimizing the number of expensive cache misses that may occur. In addition, as each node traverses the vector, the node will prefetch the data for the next packet in the vector, thereby hiding the memory latency and associated read-data dependencies.

The ability to split packet processing into a graph of nodes consequently enables pipelined execution for VPP. On multi-core systems, processing performance can improve by allowing multiple threads to exercise the same node in the graph. Alternatively, the graph can split across CPU cores, forming a threaded-pipeline model.

An implementation of a network program in VPP consists of one or more nodes that process vectors of packets, where each node contains the code for a logical stage in the program. Since there is no boundary restriction on logic contained in each node, there is a high variability in deciding how a program is divided into VPP nodes. Each implementation chooses the division that optimizes some parameter, such as cache locality.

The current VPP codebase contains a set of highly-efficient hand-tuned implementations of common network protocols and features. These correspond to a collection of packet processing nodes that are loaded by default as the standard VPP installation starts up. In these implementations, a distinctive pattern, typical to most nodes, is the unrolling of instructions to process multiple packets in a single iteration of the vector-traversal loop. Packets processed in the same iteration are operated on near-simultaneously by interleaving the instructions relevant to each packet before moving on to the next instruction. This design is to maximize the utilization of CPU registers, since each packet is only likely to require the use of part of the registers available on the processor. Memory locality and instruction cache hits are also improved, as processing multiple packets in parallel are very likely to perform the same set of instructions and refer to similar memory locations.

VPP also supports adding custom packet processing logic through the use of VPP *plugins*. VPP plugins provide modularity and extensibility in the packet processing pipeline by allowing users to add new nodes with custom packet processing logic. A plugin has the full freedom to introduce new nodes and rearrange or delete existing nodes in the graph. A plugin is even able to substitute nodes responsible for packet input and output from the device, thereby potentially supplanting the entire processing graph with nodes introduced by the plugin. VPP plugins are typically developed and compiled separately from each other and the core VPP code. An example of VPP graph structure with is shown in Figure 1.

VPP has demonstrated outstanding performance in the software switch class with its existing network implementations. In particular, it has achieved line-rate performance when running common forwarding programs on commodity x86 CPUs [13]. It also attains better performance comparisons against other software switches such as OVS [12]. The efficiencies that make these results possible are only

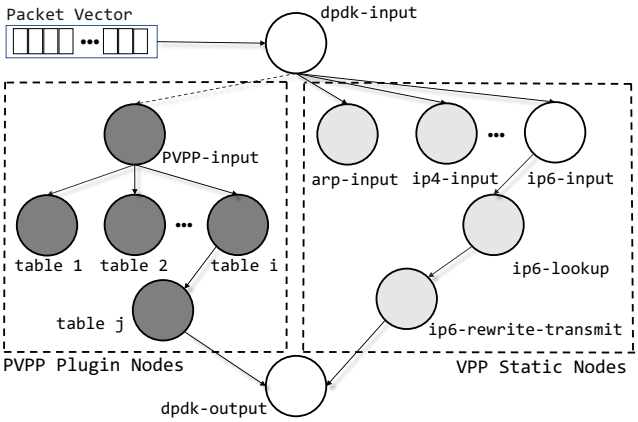


Figure 1: VPP node graph structure with PVPP plugin.

through manipulating the underlying low-level architecture in ways such as processing multiple packets simultaneously in a single packet vector iteration. The cost of operating on packets at low-level, however, is that it is difficult to produce correct implementations of packet forwarding logic. A programmer who wishes to produce efficient implementations in VPP must possess specialized knowledge in the fields of networks, operating systems, memory management and compiler optimization techniques, among others. The hand-written implementations, such as those that currently exist in the VPP code base, are difficult to evolve as the underlying processor evolves. An increase in number of registers available, for example, could potentially improve performance, but existing code that is hand-tuned for a set number of registers must also be retrofitted manually in order to take advantage of this change.

Ideally, we would like to automatically leverage the low-level interfaces available in VPP through the use of a compiler. This would show that it is possible for a software switch to expose its underlying architecture, bringing performance and other benefits, while not requiring the programmer to possess the knowledge need for an efficient implementation. To achieve this goal, we present our work-in-progress, PVPP.

3 PVPP ARCHITECTURE

We now discuss PVPP's architecture in detail. We first discuss methods for embedding P4 functionality as a VPP plugin. Then, we discuss the design choices and optimizations implemented in P4-to-PVPP compiler that generates the plugin.

3.1 Separate Compilation via VPP Plugin

PVPP integrates an input P4 program into VPP as a plugin. Embedding the entire P4-related logic into a plugin has many advantages. One major advantage is that compiling a new P4 program as a different VPP plugin does not require recompilation of the whole VPP framework, as plugins are designed to compile and load separately. This is a clear advantage over the approach taken by PISCES [10]. PISCES requires recompilation of the entire OVS codebase for each new P4 program, since the code generated by the P4-to-OVS compiler is linked statically and compiled with the whole OVS baseline source code. Another advantage is in the distribution of compiled code to multiple target systems. A plugin is compiled only once before it is distributed to different VPP targets, even if those targets are running different versions of VPP or configurations.

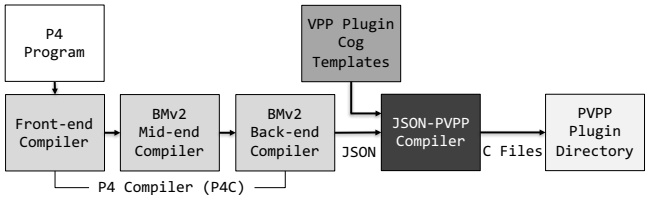


Figure 2: PVPP compiler architecture, consisting of two parts: P4C and JSON to PVPP.

3.2 P4-to-PVPP Compiler

The compilation of P4-to-PVPP code is a two step process. The first step runs the open-source P4 compiler (P4C) [3], which takes the input P4 program and generates an intermediate representation in JavaScript Object Notation (JSON) format. The overall structure of P4C is shown in the left half of Figure 2. In the second step, we feed this intermediate representation to our JSON-to-VPP compiler to generate a customized VPP plugin with single-node or multiple-node implementations. The single-node implementation creates a plugin with only one node that performs the entire parse, match, and action logic defined in the input P4 program. In contrast, the multiple-node implementation creates a plugin with a set of nodes that are each responsible for a subset of the parse, match, and action logic. In our initial multiple-node implementation, PVPP creates a separate node corresponding to each P4 table, where the node performs match and actions for that table. The overall process of compiling JSON to PVPP is shown in the right half of Figure 2.

Stage 1: P4 to JSON.. P4C first transforms the input P4 program into an internal intermediate representation [4] by running it through the front-end compiler of P4C. The front-end compiler is target-independent and is mainly responsible for P4 program validation, type checking, and performing target-independent optimizations. Given the intermediate output, P4C runs it through the target-dependent mid-end

and back-end to perform optimizations driven by target-dependent policies and generates an optimized final representation. This final representation can be in various forms, such as code snippets or data objects; in PVPP's case, JSON output is generated.

Stage 2: JSON to PVPP.. Given the JSON output from P4C, the custom Python-based compiler generates all the necessary C code to build a VPP plugin. Given the JSON representation, we utilize the Cog [1] template engine to read a set of template files to find specially marked sections used to generate code, and write the final output to a file. The output is merged with other static files to build a VPP plugin.

4 PVPP COMPILER OPTIMIZATIONS

We discuss the compiler optimizations that exploit the low-level details that are accessible in VPP plugins.

Reducing metadata access and pointer dereferences. One of the most important compiler optimization is to implement an efficient memory management scheme. To do so, the compiled code is designed so that memory dereferences are eliminated whenever possible. One place where this is evident is in the storage of metadata. A good approach we found to optimize metadata storages is to flatten and merge all P4 metadata structures into a single C structure. Although this is wasteful in space, as not all types of metadata will be valid for every possible path of the packet processing pipeline, we found significant performance improvements compared to handling metadata at multiple locations that a more direct translation of P4 declarations would produce.

Similarly, PVPP represents all action-table relationships as pre-computed pointers. PVPP determines the next table by the action taken according to the matched rule of the current table, it would know in advance the pointers to tables and actions at compile time. Therefore, we can store the relevant pointers in a rule at the time it is inserted into a table, reducing the amount of pointer arithmetic needed during rule matching stage.

Reducing number of tables and metadata. Naïvely, P4C produces a final JSON output containing a number of redundant tables and metadata. For example, P4C generates a redundant table with no match rule for performing interface output selection. Such redundant tables result in extraneous loops of table- and action-lookups for our single-node implementation, and they result in iterations through the extraneous set of nodes in the multiple-node implementation. Thus, we add an optimization to remove redundant tables and metadata that do not serve any VPP-specific function in the JSON output, prior to feeding the output to the final stage of the compiler.

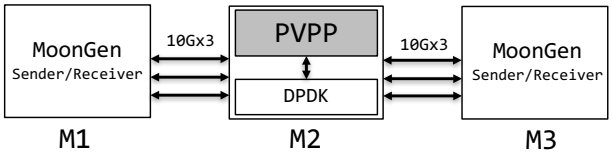


Figure 3: Topology of our experimental setup.

Multiple-node pipeline implementation. The most unique characteristic of the VPP framework is that packets traverse through a graph of processing nodes. Unfortunately, finding the best division of tasks in a packet processing pipeline into a set of nodes in a directed graph is non trivial. We add an optimization to allow users to annotate the P4 program to specify whether they want to create a plugin with a single node or multiple nodes. For a match-action pipeline defined by a P4 program, the obvious division we implement is to separate each table into different nodes. However, evaluation shows that multiple-node implementation performs slightly worse than the single-node implementation for our benchmark programs. We discuss the reasons behind this performance overhead and other methods for optimizing the division of tasks in the next section.

5 EVALUATION

We now discuss preliminary performance and scalability results using our benchmark application. Then, we discuss the comparison results between PVPP and PISCES.

5.1 Experimental Setup

Figure 3 shows the topology of our setup for evaluating the forwarding performance of PVPP. We use three PowerEdge R730xd servers with two 8-core, 16-thread Intel Xeon E5-2640 v3 2.6GHz CPUs running the Proxmox Virtual Environment [9] Kernel version 4.2.6-1-pve, an open-source server virtualization platform that uses virtual switches to connect VMs. These machines are equipped with one dual-port and one quad-port Intel X710 10 Gbps NIC. We configure two of these machines, namely `M1` and `M3`, with MoonGen [7], for sending and receiving 64-byte packets, respectively, at 14.88 million packets per second (Mpps). We connect these six interfaces to a third machine, `M2`, running PVPP, causing `M2` to process a maximum of 60 Gbps of traffic.

5.2 Baseline End-to-End Performance

We first measure the throughput of our benchmark application to establish the baseline performance with no optimization. The overview of the application is shown in Figure 4. In this application, the packets received at ingress are processed through PVPP's parser, allocating memory addresses for the IP and Ethernet headers. Then, the `IPv4_match` table

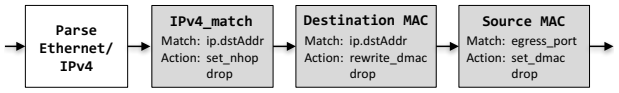


Figure 4: Control flow of our benchmark application. White boxes correspond to parsers and grey boxes correspond to tables.

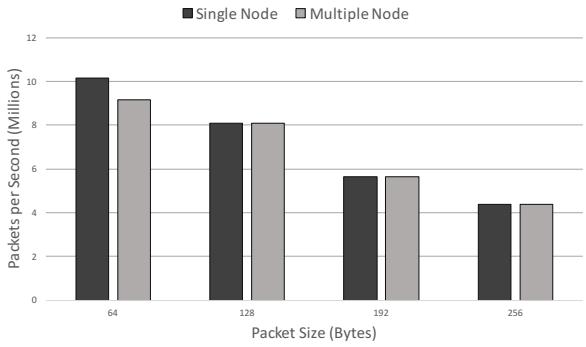
matches on the destination IP address of the packet, performs a TTL decrement, and writes the next hop IP address and the egress port based on the installed rules. The packet then goes through two separate tables rewriting the destination and source MAC address. Finally, the packet is sent out to the interface connected to M3. First line of Table 1 shows the baseline throughput for the given benchmark application.

5.3 Optimized End-to-End Performance

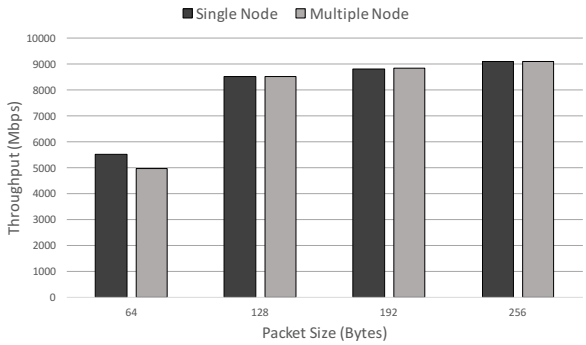
We now show the effects of the optimizations that we discussed in Section 4. The measurements are obtained from running the same benchmark application described in Section 5.2. Table 1 shows the incremental improvement in throughput for each of the optimizations. We see the greatest improvements when reducing the number of match-action tables, which has a similar effect of removing a processing node in the multiple-node implementation. Reducing the number of pointer dereferences provides further improvements, since pointer dereferencing requires a large number of CPU cycles.

One interesting observation to emphasize is the multiple packet processing optimization for the single-node implementation. Recall in Section 2, we observed that VPP nodes often processed multiple packets in parallel within a single vector traversal loop iteration. We implemented the same optimization, such that the compiler is able to generate a target network program that automatically unrolls the loop and process two packets during a single iteration. Interestingly, this optimization in the single-node implementation did not show any improvement. We hypothesize that this is the result of a long series of operations that a packet must go through when the entire pipeline is fit into one node, thus requiring a large number of registers for processing each packet. Also, the single-node implementation cannot fully reap the benefits of memory locality or i-cache hits because the packets have to be processed through the entire pipeline—straining the cache—before processing the next packet.

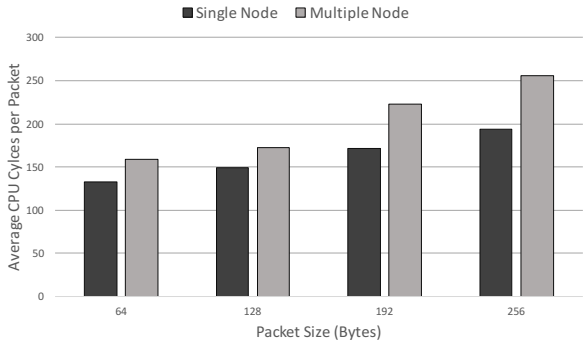
With all the optimizations, PVPP's throughput is 10.209 Mpps for the single-node implementation and 9.197 Mpps for the multiple-node implementation. The throughput of vanilla VPP for the equivalent application is 10.748 Mpps,



(a) Forwarding performance in Mpps.



(b) Forwarding performance in Mbps.



(c) Number of CPU cycles consumed per packet.

Figure 5: Forwarding performance of PVPP for the benchmark application across one 10Gbps interface.

which is 5.25% higher than the single-node implementation and 16.9% higher than the multiple-node implementation.

Figure 5 shows the detailed end-to-end performance of optimized PVPP for our benchmark application. Note that the measurement for CPU cycles only includes the cycles that were spent on the packet processing and does not include the cycles spent for DPDK interactions and packet output. The experiment shows an overhead of about 10% for both the Mpps and Mbps measurements for a 64-byte packet and

Optimization	Single-Node (Mpps)	Single-Node Increment (%)	Multiple-Node (Mpps)	Multiple-Node Increment (%)
Unoptimized	7.860	N/A	7.051	N/A
Removing Redundant Tables	9.248	+1.388 (+17.7%)	8.381	+1.330 (+18.9%)
Reducing Metadata Access	9.508	+0.260 (+2.81%)	8.501	+0.120 (+1.43%)
Multiple Packet Processing	9.508	+0.000 (+0.00%)	8.800	+0.299 (+3.52%)
Reducing Pointer Dereferences	10.008	+0.500 (+5.26%)	9.023	+0.223 (+2.53%)
Caching Interface Mapping	10.209	+0.201 (+2.01%)	9.197	+0.174 (+1.93%)

Table 1: Incremental improvements of each optimizations for PVPP

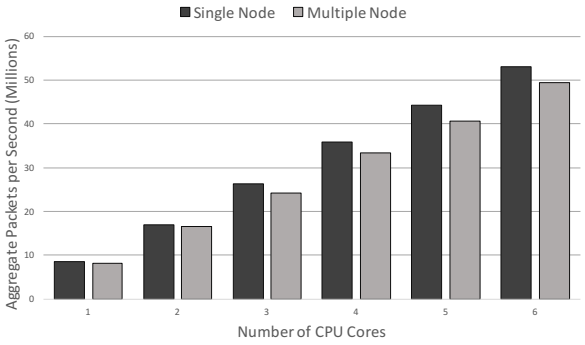


Figure 6: Effect of the number of cores used for PVPP to throughput (Mbps). Effect on Mpps is the same. Mpps = Mbps/(8 * 64).

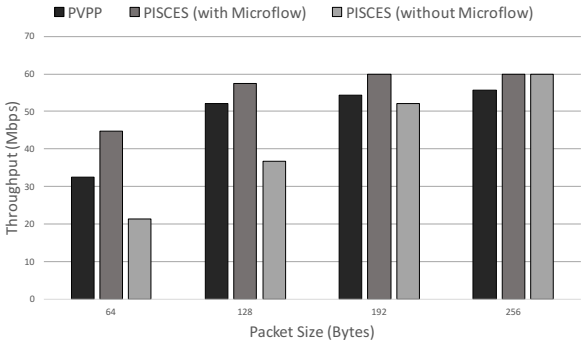


Figure 7: Throughput comparison between the single-node implementation of PVPP and PISCES with and without microflow cache.

about 30% overhead for the number of CPU cycles spent across all the packet sizes for the multiple-node implementation versus the single-node implementation. This overhead is mainly the result of additional VPP related operations for moving packets between different nodes.

Another interesting observation is the throughput versus CPU cycles. The measurements show that the increase in the CPU cycles spent per packet with the larger packets are greater for multiple-node implementation versus single-node implementation. However, the throughput measurements of two implementations counter-intuitively show a comparable increase. This result is mainly due to the fact that PVPP becomes I/O-bound rather than CPU-bound for larger packet sizes, thus the increase in CPU cycles spent are not reflected in the throughput measurements.

5.4 Multi-Core Performance

We also measure the throughput of our benchmark application while varying the number of CPU cores and the number of input interfaces used by PVPP, to quantify the scalability of throughput versus the number of CPUs used. Only one CPU processes the packet from one designated interface.

CPU Cycles	PVPP	PISCES (with microflow)	PISCES (no microflow)
End-to-End	132.9	100.6	166.0

Table 2: Average number of CPU cycles consumed for processing a 64-byte packet in PVPP and PISCES with and without microflow cache.

For this experiment, we started bidirectional traffic between the three interface pairs, generating 64-byte sized packets at the maximum rate of 60 Gbps. This results in packets flowing through six ingress ports that PVPP must process. The procedure of adding an interface accompanies launching another VPP thread on a new CPU core that processes the packets arriving at the newly added ingress interface. Figure 6 shows the relationship between the performance of PVPP with varying number of CPU cores. We observe that throughput increases linearly with the addition of CPU cores.

5.5 Comparing PVPP and PISCES

We compare the forwarding performance between PVPP and PISCES [10], with and without micro-flow cache turned on. PISCES underlying switch target, OVS, relies on caches to achieve good forwarding performance. The primary OVS cache is its mega-flow cache. With the mega-flow cache, OVS can combine the results of tables that a packet visits in the match-action pipeline into a single flow rule by (lazily) computing the cross-product of the tables [8]. This rule is then installed in the mega-flow cache. Mega-flow cache wildcard matches on headers fields and, hence, can still have a significant toll on performance. Thus, OVS also includes a micro-flow cache, an exact-match cache, which is a maps from a packet's five-tuple to a mega-flow cache entry.

For comparison, we used a simple (switch) program that matches on the destination MAC and sends the packet to the appropriate egress interface. We send various sized packets to all six interfaces using six CPU cores, each pinned to a distinct ingress interface, resulting in a total of 60 Gbps traffic.

Figure 7 shows the comparisons. PVPP performs comparably with PISCES with micro-flow cache enabled, and better with micro-flow cache disabled. Table 2 shows the comparisons of end-to-end CPU cycles with different cache configurations. A caveat to note is that this is not an exact comparison, because of the difference in implementations of the two switches. Nevertheless, CPU cycles spent per packet are comparable, confirming the similarities in throughputs.

6 DISCUSSION AND FUTURE WORK

PVPP presents a promising argument that low-level interfaces available in a software switch can be utilized to generate a more efficient P4-programmed switch. In order to further strengthen the argument and to provide a more functional P4 programmable software switch, we are planning to complement our work with the following action items.

Automated and optimal node splits. Current scheme for generating multiple nodes for a PVPP plugin is to let the compiler split a P4 program by creating a node for each P4 table. Our evaluation shows this method can lead to worse performance. As part of our future work, we intend to create schemes using analysis of input P4 programs to generate more intelligent node splits for better performance.

Handling multiple packets. As mentioned in Sections 2 and 5, processing two packets per loop iteration is often beneficial. We believe that further unrolling may yield better performance, particularly for match-action tables with a small number of instructions. PVPP provides the opportunity to explore this space by developing a P4 program analyzer that determines the optimal number of packets to process in parallel.

Extending P4 feature support. PVPP currently lacks some features that P4 supports such as data plane states (i.e. registers, counters or meters). One advantage of VPP is the lack of OVS-like cache structures that enables relatively straightforward implementation of the data plane states. One possible design suggestion to implement data plane states in PVPP is to compile all match-action tables that refer the stateful features into a single node, with the state allocated and referenced from only within that node. This approach avoids adding costly locks that are needed if more than one node references the same state. However, supporting parallel accesses on the same states across multiple nodes require a more sophisticated design to ensure correct stateful operations.

REFERENCES

[1] Cog. http://nedbatchelder.com/code/cog/.
[2] DPDK: Data Plane Development Kit. http://dpdk.org.
[3] P4 Compiler (P4C). https://github.com/p4lang/p4c-bm.
[4] P4 Intermediate Representation. https://github.com/p4lang/p4c/.
[5] A. Bianco, R. Birke, L. Giraudo, and M. Palacin. OpenFlow Switching: Data Plane Performance. In *IEEE International Conference on Communications (ICC)*, 2010.
[6] 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. *ACM SIGCOMM Computer Communication Review (CCR)*, July 2014.
[7] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *ACM The Internet Measurement Conference (IMC)*, 2015.
[8] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The Design and Implementation of Open vSwitch. In *USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2015.
[9] Proxmox Virtual Environment. https://www.proxmox.com.
[10] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford. Pisces: A programmable, protocol-independent software switch. In *ACM SIGCOMM*, 2016.
[11] Vector Packet Processing (VPP) Platform. https://fd.io.
[12] Validating Cisco's NFV Infrastructure Pt. 1. http://www.lightreading.com/nfv/nfv-tests-and-trials/validating-ciscos-nfv-infrastructure-pt-1/d/d-id/718684.
[13] VPP Performance Tests. https://docs.fd.io/csit/rls1701/report/vpp_performance_tests/.