PISCES: A Programmable, Protocol-Independent Software Switch

Muhammad Shahbaz[⋆], Sean Choi[⋄], Ben Pfaff[†], Changhoon Kim[‡],
Nick Feamster[⋆], Nick McKeown[⋄], Jennifer Rexford[⋆]
[⋆]Princeton University [⋄]Stanford University [†]VMware, Inc [‡]Barefoot Networks, Inc

Abstract

Virtualized data-centers use software hypervisor switches to steer packets to and from virtual machines (VMs). The switch frequently needs upgrading and customization—to support new protocol headers or encapsulations for tunneling or overlays, to improve measurement and debugging features, and even to add middlebox-like functions. Software switches are typically based on a large body of code, including kernel code. Changing the switch is a formidable undertaking requiring domain mastery of network protocol design *and* developing, testing, and maintaining a large, complex code-base. In this paper, we argue that changing how a software switch forwards packets should not require intimate knowledge of its implementation. Instead, it should be possible to specify how packets are processed and forwarded in a high-level domain-specific language (DSL) such as P4, then compiled down to run on the underlying software switch. We present PISCES, a software switch that is not hard-wired to specific protocols, which eases adding new features. We also show how the compiler can analyze the high-level specification to optimize forwarding performance. Our evaluation shows that PISCES performs comparably to Open vSwitch, a hardwired hypervisor switch, and that PISCES programs are about 40 times shorter than equivalent Open vSwitch programs.

1 Introduction

When we think of Ethernet switches, we tend to think of a physical box in a wiring closet or sitting at the top of a rack of servers. But every hypervisor in a virtualized data center contains a software switch, such as Open vSwitch [39], to connect VMs to the outside world. Because such a data center contains more hypervisors than hardware switches, it contains more software switches than physical ones. Likewise, because each hypervisor hosts several VMs, the data center has more virtual Ethernet ports than physical ones. Thus, the design of software switches is an important topic.

One of the main advantages of a software hypervisor switch is that it can be upgraded more easily than a hardware switch. As a result, hypervisor switches are frequently called upon to support new tunneling and overlay protocols, improved troubleshooting and debugging features, and middlebox functions such as load balancing, address virtualization, and encryption. In the future, as data center owners customize and optimize their infrastructure, many more features will be added to the hypervisor switch.

Each new feature requires the hypervisor switch software to be changed, yet customizing a hypervisor switch is more difficult than it may appear. First, customization is technically challenging because most of the machinery that enables fast packet forwarding resides in the kernel. Writing kernel code requires domain expertise that most network operators lack, and thus introduces a significant barrier for developing and deploying new features. Some steps have been made to accelerate packet forwarding in user-space (*e.g.*, DPDK [24] and Netmap [43]), yet still it requires significant software development expertise and intimate familiarity with a large, intricate, and complex code-base. The available options for customizing a hypervisor switch thus require both domain mastery of network protocol design *and* the ability to develop, test, and maintain a large, complex code-base. Furthermore, customization requires not only incorporating changes into switch code, but also *maintaining* these customizations—a tall order in enterprises and organizations that do not have large software development teams.

Changing how a software switch forwards packets should not need intimate knowledge of how the switch is implemented. Rather, it should be possible to specify custom network protocols in a domain-specific language (DSL) such as P4 [8], which is then compiled to custom code for the hypervisor switch. Such a DSL would support customizing the forwarding behavior of the switch, without requiring changes to the underlying switch implementation. Decoupling custom protocol implementations from underlying switch code also makes it easier to maintain these customizations, since they remain independent of the underlying switch implementation. With a standardized DSL, customizations may also be more easily portable from one switch to another, in software or hardware, that supports the same language.

A key insight, borrowed from a similar trend in hardware switches [9, 37], is that the underlying switch should be a substrate, well-tuned to process packets at high speed, but not tied to a specific protocol. In the extreme, the switch is said to be "protocol independent" meaning that before it is told how to process packets (via a DSL), it does not know what a protocol is. Put another way, protocols are represented by programs written in the DSL, which protocol authors create.

Our approach in software is similar. We assume the DSL specifies which packet headers to parse and the structure of the match-action tables (*i.e.*, which header fields to match and which actions to perform on matching headers). The underlying software substrate is a generic engine, optimized to parse, match, and act upon the packet headers in the form the DSL specifies.

Expressing these customizations in a DSL, however, entails compilation from the DSL to code that runs in the underlying switch. Compared to a switch that is handwritten to implement fixed protocols, this protocol compilation process may reduce the efficiency of the underlying implementation and thus come at the cost of performance. Our goals in this paper are to (1) quantify the additional cost that expressing custom protocols in such a DSL produces; and (2) design and evaluate domain-specific compiler optimizations that reduce the performance overhead as much as possible. Ultimately, we demonstrate that, with the appropriate compiler optimizations, the performance of a *protocol-independent* software switch—a switch that supports custom protocol specification in a high-level DSL without direct modifications to the low-level source code—approaches parity with the native hypervisor software switch.

PISCES is the first software switch that allows custom protocol specification in a high-level DSL, without requiring direct modifications to switch source code. Our results are promising, particularly given that Open vSwitch, our base code, was not designed to support protocol independence. Nevertheless, our results demonstrate that the "cost of programmability" in hypervisor switches is negligible. We expect our results will inspire the design of new protocol-independent software switches running at even higher speeds.

We first motivate the need for a customizable hypervisor software switch with a description of real use cases from operational networks (Section 2) and present background information on both P4 and Open vSwitch (Section 3). We then present PISCES, a prototype protocol-independent software switch based on Open vSwitch that can be programmed from a high-level DSL such as P4. The prototype uses a set of domain-specific compiler optimizations to reduce the performance overhead of customizing the software switch using a DSL (Section 4). Finally, we evaluate code complexity and forwarding performance of PISCES (Section 5). Our evaluation shows that PISCES programs are on average about 40 times shorter than equivalent OVS programs and incur a forwarding performance overhead of only about 2%.[1]

2 The Need for a Protocol-Independent Switch

We say that PISCES is a *protocol-independent* software switch, because it does not know what a protocol is, or how to process packets on behalf of a protocol, until the programmer specifies it. For example, if we want PISCES to process IPv4 packets, then we need to describe how IPv4 packets are processed in a P4 program. In the P4 program (*e.g.*, `IPv4.p4`) we need to describe the format and fields of the IPv4 header, including the IP addresses, protocol ID, TTL, checksum, flags, and so forth. We also need to specify that we use a lookup table to store IPv4 prefixes, and that we search for the longest matching prefix. We also need to describe how a TTL is decremented, a checksum is updated and so on. The P4 program captures the entire packet processing pipeline, which is compiled to source code for OVS that specifies the switch's match, action, and parse capabilities.

A protocol-independent switch brings many benefits:

Adding new standard or private protocol headers. New protocol headers are being standardized all the time, particularly for data centers. In recent years, VXLAN, NVGRE, Geneve have all been standardized, and STT and NSH are also being discussed as potential standards. Private, proprietary protocols are also added, to provide a competitive advantage by, for example, creating better isolation between applications, or by introducing novel congestion marking. Before new protocols can be deployed, all hardware and software switches must be upgraded to recognize the headers and process them correctly. For hardware switches, the data center owner must provide requirements to their chip vendor and then wait three to four years for the new feature to arrive, if the vendor agrees to add the feature at all. In the case of software switches, they must wait for the next major revision, testing, and deployment cycle. Even modifying an open-source software switch is not a panacea because once the data center owner directly modifies the open-source software switches to add their own custom protocols, these modifications still need to be maintained and synchronized with the mainline codebase, introducing significant code maintenance overhead as the original open-source switch continues to evolve. If instead the data-center owner could add new protocols to their P4 program, they would be able to compile and deploy the new protocols much more quickly.

Removing a standard protocol header. Data-center networks typically run a lot *fewer* protocols than legacy campus and enterprise networks, in part because most of the traffic is machine-to-machine and many legacy protocols are not needed (*e.g.*, multicast, RSVP, L2-learning). It therefore benefits the data-center owner to remove unused protocols, thus eliminating the burden of needing to understand unused protocols and the CPU cycles required to parse them. It is bad enough to have to support many protocols; much worse to have to support and understand protocols that are not even used in your network. Therefore, data-center owners frequently want to eliminate unused protocols from their switches, NICs, and operating systems.[2] Removing protocols from conventional switches is difficult; for hardware, it means waiting for new silicon, and for software switches it means

[1]Reproducibility: The final version of the paper will be accompanied by an open-source version of PISCES, the compiler, the patches to OVS, and the P4 programs needed to reproduce our results.

[2]For example, AWS reportedly only forwards packets using IPv4 headers.

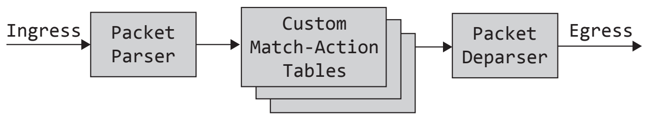


Figure 1: *P4 abstract forwarding model.*

wrestling with a large codebase to extract a specific protocol. In PISCES, removing an unused protocol is as simple as removing (or commenting out) unused portions of a protocol specification and recompiling the switch source code. (Section 5.2.2 shows how this can even improve performance.)

Adding better visibility. As data-centers get larger and are used by more applications, it becomes important to understand the network's behavior and operating conditions. Failures can lead to huge loss in revenue, exacerbated by long debug times as the network gets bigger and more complicated. It is therefore not surprising that there is growing interest in making it easier to see what the network is doing. Improving network visibility might entail adding new counters, generating new probe packets, or adding new protocols and actions to collect switch state (as is enabled by in-band network telemetry [28]). Users will want to see how queues are evolving, latencies are varying, whether tunnels are correctly terminated, and whether links are still up. Often, during an emergency, users want to add visibility features very quickly. Having them ready to deploy, or being able to modify forwarding and monitoring logic quickly may reduce the time to diagnose and fix a network outage.

Adding entirely new features. If users and network owners can modify the forwarding behavior, they may even add entirely new features. For example, over time we can expect switches to take on more complex routing, such as path-utilization aware routing [2], new congestion control mechanisms [6, 15, 27], source-controlled routing [40], new load-balancing algorithms [21], new methods to mitigate DDoS [3, 20], and new virtual-to-physical gateway functions [13]. If a network owner can upgrade infrastructure to achieve greater utilization or more control, then they will know best how to do it. Given the means to upgrade a program written in a DSL like P4 for adding new features to a switch, we can expect network owners to improve their networks much more rapidly.

3 Background

PISCES is a *software switch* whose forwarding behavior is specified using a *domain-specific language*. Our prototype PISCES switch is based on the Open vSwitch (OVS) software switch and is configured using the P4 domain-specific language. We describe both P4 and OVS below.

Domain-Specific Language: P4. P4 is a domain-specific language that expresses how the pipeline of a network forwarding element should process packets using the abstract forwarding model shown in Figure 1. In this model, each packet first passes through a programmable *parser*, which extracts headers. The P4 program specifies the structure of each possible header as well as a parse graph that expresses ordering and dependencies. Then, the packet passes through a series of match-action tables (MATs). The P4 program specifies the fields that each of these MATs may match and the control flow among them, as well as the spectrum of permissible actions for each table. At "runtime" (*i.e.*, while the switch is forwarding packets), controller software may add, remove, and modify table entries with particular match-action rules that conform to the P4 program's specification. Finally, a *deparser* writes the header fields back onto the packet before sending it out the appropriate port.

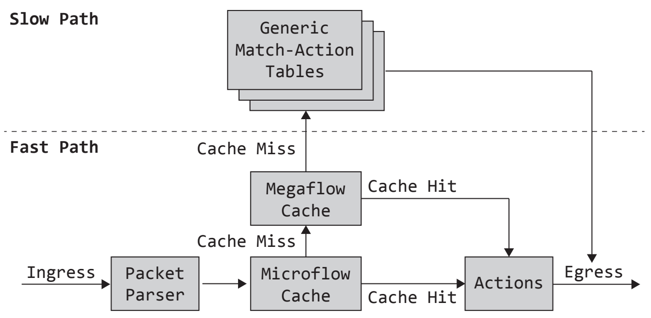


Figure 2: *OVS forwarding model.*

Software Switch: Open vSwitch. Open vSwitch (OVS) is widely used in virtualized data-center environments as a software switch running inside the hypervisor. In such an environment, OVS switches packets among virtual interfaces to VMs and physical interfaces. OVS implements common protocols such as Ethernet, GRE, and IPv4, as well as newer protocols found in datacenters, such as VXLAN, Geneve, and STT for virtual network overlays.

The Open vSwitch virtual switch is divided into two important pieces, called the *slow path* and the *fast path* (aka *datapath*), as shown in Figure 2. The slow path, which is the larger component of the two, is a user-space program; it supplies most of the intelligence of Open vSwitch. The fast path acts as a caching layer that contains only the code needed to achieve maximum performance. Notably, the fast path must pass any packet that misses its caches to the slow path to get instructions for further processing. Open vSwitch includes a single, portable slow path and multiple fast-path implementations for different environments: one based on a Linux kernel module, another based on a Windows kernel module, and another based on Intel DPDK [24] userspace forwarding. The DPDK fast path yields the highest performance, so we use it for our work; with additional effort, our work could be extended to the other fast paths.

As an SDN switch, Open vSwitch relies on instructions from a controller to determine its behavior, specifically using the OpenFlow protocol [34]. OpenFlow specifies behavior in terms of a collection of *match-action tables*, each of which contains a number of entries called *flows*. In turn, a flow consists of a *match*, in terms of packet headers and metadata, *actions* that instruct the switch what to do when the match evaluates to true, and a numerical *priority*. When a packet arrives at a particular match-action table, the switch finds a matching flow and executes its actions; if more than one flow matches the packet, then the flow with the highest *priority* takes precedence.

A software switch that implements the behavior exactly as described above cannot achieve high performance, because OpenFlow packets often pass through several match-action tables, each of which requires general-purpose packet classification. Thus, Open vSwitch relies on caches to achieve good forwarding performance. The primary OVS cache is its *megaflow cache*, which is structured much like an OpenFlow table. The idea behind the megaflow cache is that one could, in theory, combine all of the match-action tables that a packet visits while traversing the OpenFlow pipeline into a single table by computing their cross-product. This is infeasible, however, because the cross-product of k tables with $n_1,\ldots,n_k$ flows might have as many as $n_1 \times \cdots \times n_k$ flows. The megaflow cache functions somewhat like a lazily computed cross-product: when a packet arrives that does not match any existing megaflow cache entry, the slow path computes a new entry, which corresponds to one row in the theoretical cross-product, and inserts it into the cache. Open vSwitch uses a number of techniques to improve megaflow cache performance and hit rate [39].

When a packet hits in the megaflow cache, the switch can process it significantly faster than the round trip from the fast path to the slow path that a cache miss would require. As a general-purpose packet classification step, however, a megaflow cache lookup still has a significant cost. Thus, Open vSwitch fast path implementations also include a microflow cache. In the DPDK fast path, the microflow cache is a hash table that maps from a packet five-tuple to a megaflow cache entry. At ingress, the fast path uses this hash table as a hint to determine which megaflow cache entry to check first. If that megaflow cache entry matches, the lookup cost is just that of the single hash table lookup. Since this is generally much cheaper than general packet classification, it is a significant optimization for traffic patterns with relatively long, steady streams of packets. (If the megaflow cache entry check fails, *e.g.*, because the flow table does not treat all packets with the same five-tuple alike, the packet continues through the usual megaflow cache lookup process, skipping the entry that it has already checked.)

4 PISCES Prototype

Our PISCES prototype is a modified version of OVS with the *parse*, *match* and *action* code replaced by C code generated by our P4 compiler. The workflow is as follows: First, the programmer creates a P4 program and uses the PISCES P4 compiler to generate new *parse*, *match* and *action* code for OVS. Second, OVS is compiled (using the regular C compiler) to create a protocol-*de*pendent switch that processes packets as described in the P4 program. To modify a protocol, a user modifies the P4 program which compiles to a new hypervisor switch binary.

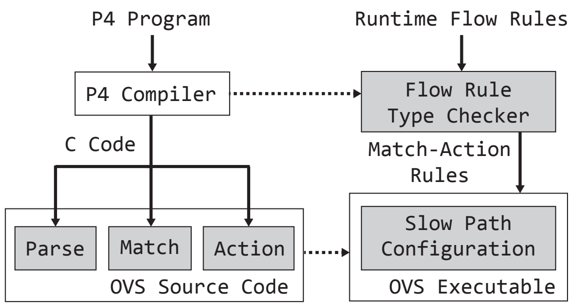


Figure 3: *The PISCES P4-to-OVS Compiler.*

We use OVS as the basis for PISCES because it is widely used and contains some basic scaffolding for a programmable switch, thus allowing us to focus only on the parts of the switch that need to be customized (*i.e.*, parse, macth, and action). The code is well-structured, lending itself to modification, and test environments already exist. It also allows for apples-to-apples comparisons: We can compare the number of lines of code in OVS to the resulting PISCES code, and we can also compare their performance.

4.1 The PISCES P4-to-OVS Compiler

P4 compilers have two parts: a front end that turns the P4 code into a target-independent intermediate representation (IR), and a back end that maps the IR to the target. In our case, the back end optimizes CPU time, latency, or other objectives by manipulating the IR, and then generates C code that replaces the parsing, match, and action code in OVS, as shown in Figure 3. The P4-to-OVS compiler outputs C source code that implements everything needed to compile the corresponding switch executable. The compilation process also generates an independent type checking program that the executable uses to ensure that any runtime configuration directives from the controller (*e.g.*, insertion of flow rules) conforms to the protocol specified in the P4 program.

Parse. The C code that replaces the original OVS *parser* is created by replacing `struct flow`, the C structure that OVS uses to track protocol header fields, to include a member for each field specified by the P4 program, and generating code to extract header fields from a packet into `struct flow`.

Match. OVS uses a general-purpose classifier data structure, based on tuple-space search [46], to implement matching. To perform custom matches, we do not need to modify this data structure or the code that manages it. Rather, the control plane can simply populate the classifier with new packet header fields at runtime, thereby automatically making those fields available for packet matching.

Action. The back end of our compiler supports custom actions by automatically generating code that we statically compile into the OVS binary. Custom actions can execute either in the OVS slow path or the fast path; the compiler determines where a particular action will run to ensure that the switch performs the actions efficiently. Certain actions (*e.g.*, set_field) can execute in either component. The programmer can offer hints to the compiler as to whether slow path or fast path implementation of an action is most appropriate.

Control flow. A P4 program defines a control flow to specify the possible sequences of match-action tables that a packet will traverse. In OVS, individual flow entries added at runtime specify control flow, allowing arbitrary control flow at runtime. Therefore, our compiler back end can implement P4 control semantics without OVS changes.

Optimizing the IR. The compiler back end contains an optimizer to examine and modify the IR, so as to generate high-performance C code. For example, a P4 program may include a complete IP checksum, but the optimizer can turn this operation into an incremental IP checksum to make it faster. The compiler also performs liveness analysis on the IR [1], allowing it to coalesce and specialize the C code. The optimizer also decides when and where in the packet processing pipeline to edit packet headers. Some hardware switches postpone editing until the end of the pipeline, whereas software switches typically edit headers at each stage in the pipeline. If necessary, the optimizer converts the IR for in-line editing. We describe the optimizer in more detail in Section 4.3.

As is the case with other P4 compilers, the P4-to-OVS compiler also generates an API to interface to the match-action tables, and extends the OVS command-line tools to work with the new fields.

4.2 Modifications to OVS

We needed to make three modifications to OVS to enable it to implement the forwarding behavior described in any P4 program.

Arbitrary encapsulation and decapsulation. OVS does not support arbitrary encapsulation and decapsulation, which a P4 program might require. Each OVS fast path has its own specialized support for various fixed forms of encapsulation. The Linux kernel fast path and DPDK fast path, for example, each separately implement GRE, VXLAN, STT, and other encapsulations. The metadata required to encapsulate and decapsulate a packet for a tunnel is statically configured. The switch uses a packet's ingress port to map it to the appropriate tunnel; on egress, the packet is encapsulated in the corresponding IP header based on this static tunnel configuration. We therefore added two new primitives to OVS, add_header() and remove_header(), to perform encapsulation and decapsulation, respectively, and performs these operations in the fast path.

Conditionals based on comparison of header fields. OpenFlow directly supports only bitwise equality tests against header fields. Relational tests such as $<$ and $>$ to compare a k-bit field against a constant can be expressed as at most k flows that use bitwise equality matches. A relational test between two k-bit fields, *e.g.* $x < y$, requires $k(k+1)/2$ such flows. To simultaneously test for two such conditions that individually take n_1 and n_2 flows, one needs $n_1 \times n_2$ flows. P4 directly supports such tests, but implementing them in

Optimization	CPU Cycles	Slow Path Trips
Inline- vs. post-pipeline editing	✓	
Incremental checksum	✓	
Parser specialization	✓	
Action specialization	✓	
Action coalescing	✓	
Stateful operations	✓	✓
Stage assignment	✓	✓

Table 1: *Back-end optimizations and how they improve performance.*

OpenFlow this way is too expensive, so we added direct support for them in OVS as *conditional actions*, a kind of "if" statement for OpenFlow actions. For example, our extension allows the P4 compiler to emit an action of the form "If $x < y$, go to table 2, otherwise go to table 3."

General checksum verify/update. An IP router is supposed to verify the checksum at ingress, and recompute it at egress, and most hardware switches do it this way. Software routers often skip checksum verification on ingress to reduce CPU cycles. Instead, it just incrementally updates the checksum if it changes any fields (e.g. the TTL).[3] Currently, OVS only supports incremental checksums, but we want to support other uses of checksums in the way the programmer intended. We therefore added incremental checksum optimization, described in Section 4.3. Whether this optimization is valid depends on whether the P4 switch is acting as a forwarding element or an end host for a given packet—if it is an end host, then it must verify the checksum—so it requires hinting by the P4 programmer.

4.3 The Compiler's Back-end Optimizer

Two aspects of a software switch ultimately affect forwarding performance: (1) the per-packet cost for fast-path processing (adding 100 cycles to this cost reduces the switch's throughput by about 500 Mbps), and (2) the number of packets sent to the slow path, which takes 50+ times as many cycles as the fast path to process a packet. Table 1 lists the optimizations that we have implemented, as well as whether the optimization reduces trips to the slow path, fast path CPU cycles, or both. The rest of the section details these optimizations.

Inline editing vs. post-pipeline editing. The OVS fast path does *inline editing*, applying packet modifications immediately (the slow path does some simple optimization to avoid redundant or unnecessary modifications). If a large number of header fields are modified, removed or inserted, it can become costly to move and resize packet data on the fly. Instead, it can be more efficient to delay editing until the headers have been processed (as hardware switches typically do). The optimizer analyzes the IR to determine how many times a packet may need to be modified in the pipeline. If the value is below a certain threshold, then the optimizer performs inline editing; otherwise, it performs post-pipeline editing. We al-

[3] If the checksum was incorrect before the update, it is still incorrect afterward, and we rely on the ultimate end host to discard the packet.

low the programmer to override this heuristic using a pragma directive.

Incremental checksum. By expressing a checksum operation in terms of a high-level program description such as P4, a programmer can provide a compiler with the necessary contextual information to implement the checksum more efficiently. For example, the programmer can inform the compiler via a pragma that the checksum for each packet can be computed incrementally [32]; the optimizer can then perform liveness analysis to determine which packet header fields change, thus making re-computation of the checksum more efficient.

Parser specialization. Protocol-independent software switches can optimize the implementation of the packet parser, since a customized packet processing pipeline (as specified in a high-level language such as P4) provides specific information about which fields in the packet are modified or used as the basis for forwarding decisions. For example, a layer-2 switch that does not make forwarding decisions based on information at other layers can avoid parsing packet header fields at those layers. Specifying the forwarding behavior in a high-level language provides the compiler with information that it can use to optimize the parser.

Action specialization. The inline editing actions in the OVS fast path group together related fields that are often set at the same time. For example, OVS implements a single fast path action that sets the IPv4 source, destination, type of service, and TTL value. This is efficient when more than one of these fields is to be updated at the same time, with little marginal cost if only one is updated. IPv4 has many other fields, but the fast path cannot set any of them.

The design of this aspect of OVS required expert knowledge: its designer knew which fields were important for the fast path to be able to change. A P4 compiler does not have this kind of expert knowledge, so a design similar to the OVS fast path would require bundling every field in, say, the IPv4 header into a single action, incurring expensive checks that can significantly impede performance. Fortunately, the high-level P4 description of the match-action and control flow allows the optimizer to identify and eliminate redundant checks in the fast-path set actions, using optimizations like dead-code elimination [1]. This way, the optimizer only checks those fields in the set actions that will actually be set in the match-action control flow.

Action coalescing. By analyzing the control-flow and match-action processing in the P4 program, we can find out which fields are actually modified and can generate an efficient, single action to directly update those fields. Thus, if a rule modifies two fields, the optimizer only installs one action in OVS.

Cached field modifications. OVS does not decrement TTLs in the fast path. Instead, the slow path relies on the fact that most packets from a given source have the same TTL. Therefore, it matches on the TTL value observed in the packet that it is forwarding and emits an action that overwrites the old value with one less than that observed value. This works because the OVS designers knew the semantics of TTL and that caching this way would yield a high hit rate. However, this will not be effective with every field. Suppose, for example, that we wished to hash a set of packet headers and store the hash value somewhere in the packet. With the OVS approach, the cache would have to match on every field that contributes to the hash value. Unless all of these inputs are as predictable as TTL, this is impractical, because the cache entry would have a "hit rate" approaching zero, defeating its purpose.

The optimizer uses dataflow analysis based on the IR to help it determine whether it can safely install a match rule in the fast path cache. If it cannot, then the packet must be processed in the slow path.

Stage assignment. OVS implements *staged lookups* [39] to reduce the number of trips to the slow path. It divides fields into four groups: metadata, L2, L3, and L4, in decreasing order of their entropy (or traffic granularity). In this model, each hash table used in a tuple space search classifier is split into four hash tables, called *stages*. The first stage searches using metadata fields only, the second using metadata and L2 fields, the third using metadata, L2, and L3 fields, and the fourth stage using all fields. A lookup therefore searches each of these stages in order. If any yields no match, the overall search terminates and only the fields included in the last stage matched must be matched in the cache entry.

The optimizer uses a heuristic to try to get the benefits of staging. It analyzes the header definitions specified in the P4 program and divides the fields into groups. We augmented the P4 language to enable a user to tag each header with a relative entropy value. For example, if header h_1 has lower entropy than header h_2, the user can assign h_2 a value of 0 and h_2 a value of 1. We then sort these headers in decreasing order of their entropy value. Once the headers are sorted, we generate stages as follows: the first stage has header h_1, the second stage has headers h_1 and h_2, the third stage has headers h_1, h_2, and h_3, and so on. The total number of stages is equal to the total number of headers in a P4 program.

5 Evaluation

We compare the complexity and performance of a virtual software switch generated by PISCES with equivalent OVS native packet processing. We compare the resulting programs along two dimensions: (1) complexity, including development and deployment complexity as well as maintainability; (2) performance, by comparing packet-forwarding performance of PISCES to the same native OVS functionality.

5.1 Complexity

Complexity indicates the ease with which a program may be modified to fix defects, meet new requirements, simplify future maintenance or cope with changes in the software environment. We evaluate two categories of complexity: (1) *development complexity* of developing baseline features for a

	LOC	Methods	Method Size
Native OVS	14,535	106	137.13
ovs.p4	341	40	8.53

Table 2: *Native OVS compared to equivalent baseline functionality implemented in PISCES.*

		Files Changed	Lines Changed
Connection Label	OVS	28	411
	ovs.p4	1	5
Tunnel OAM Flag	OVS	18	170
	ovs.p4	1	6
TCP Flags	OVS	20	370
	ovs.p4	1	4

Table 3: *The number of files and lines we needed to change to implement various functionality in P4, compiled with PISCES, compared to adding the same functionality to native OVS.*

software switch; and (2) *change complexity* of maintaining an existing software switch.

5.1.1 Development complexity

We evaluate development complexity with three different metrics: lines of code, method count, and average method size. We count lines of code simply by counting line break characters and the number of methods by counting the number of subroutines in each program, as measured using `ctags`. Finally, we divide lines of code by number of methods to arrive at the average method size. A high average might indicate that (some) methods are too verbose or complex. `ovs.p4` contains the representation of the headers, parsers and actions that are currently supported in OVS [38]. Much of the code in Open vSwitch is out of the scope of P4, so our measurements include only the files that are responsible for protocol definitions and header parsing.

Table 2 summarizes each of these metrics for the native OVS header fields and parser implementation, and the equivalent logic in P4.[4] PISCES reduces the lines of code by about factor of 40 and the average method size by about a factor of 20.

5.1.2 Change complexity

To evaluate the complexity of *maintaining* a protocol-independent software switch in PISCES, we compare the effort required to add support for a new header field in a protocol that is otherwise already supported, in OVS and in P4. Table 3 shows our analysis of changes to add support for three fields: (1) *connection label*, a 128-bit custom metadata to the connection tracking interface; (2) *tunnel OAM flag*, which many networking tools use to distinguish test packets from real traffic; and (3) *TCP flags*, a modification which adds support for parsing all of the TCP flags. The numbers for Open vSwitch, based on the public Open vSwitch commits that added support for these fields [35], are conservative because they include only the changes to one of the three OVS fast path implementations.

[4]We reused the same code for the match-action tables in both implementations because this logic generalizes for both OVS and a protocol-independent switch such as PISCES.

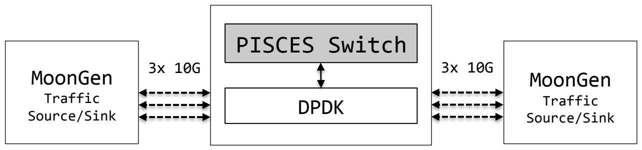


Figure 4: *Topology of our evaluation platform.*

The results demonstrate that modifying just a few lines of code in a single P4 file is sufficient to support a new field, whereas in OVS, the corresponding change often requires hundreds of lines of changes over tens of files. Among other changes, one must add the field to `struct flow`, describe properties of the field in a global table, implement a parser for the field in the slow path, and separately implement a parser in one or more of the fast paths.

5.2 Forwarding Performance

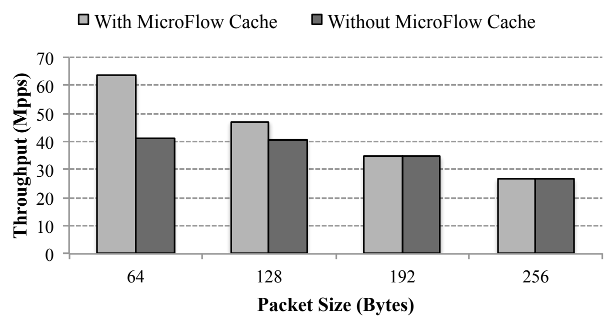
In this section, we compare OVS and PISCES packet-forwarding performance.

5.2.1 Experiment setup and evaluation metrics

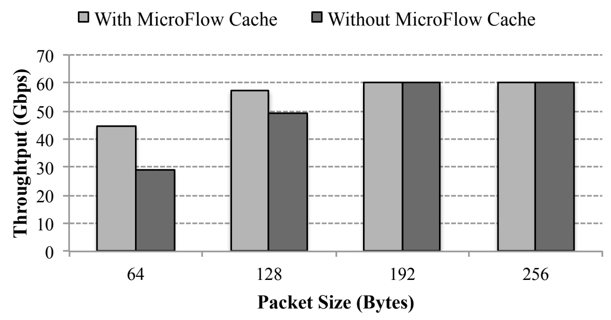
Figure 4 shows the topology of the setup for evaluating the forwarding performance of PISCES. We use three PowerEdge R730xd servers with 8-core, 16-thread Intel Xeon E5-2640 v3 2.6GHz CPUs running the Proxmox Virtual Environment [41], an open-source server virtualization platform that uses virtual switches to connect VMs, with Proxmox Kernel version 4.2.6-1-pve. Each of our machines is equipped with one dual-port and one quad-port Intel X710 10GE NIC. We configured two such machines with MoonGen [17] to send minimum-size 64-byte frames at 14.88 million packets per second (Mpps) full line rate on three of the 10GE interfaces [43], leaving the other interfaces unused. We connect these six interfaces to our third machine, the device under test, sending a total of 60 Gbps of traffic for our PISCES prototype to forward.

We consider throughput and packets-per-second to compare the forwarding performance of PISCES and OVS, using the MoonGen packet generator to generate test traffic for our experiments. To further understand performance bottlenecks, we use the machine's time-stamp counter (TSC) to measure the number of CPU cycles used by various packet processing operations (*i.e.*, parser, megaflow cache lookup, and actions). When reporting CPU cycles, we report the average CPU cycles per packet over all packets forwarded in an experiment run; each run lasts for 30 seconds and typically forwards about seven million packets.

For most of our experiments, we disabled OVS's microflow cache because it relies on matching a hash of a packet's five-tuple, which most NICs can compute directly in hardware. Although OVS's microflow cache significantly im-



(a) *Forwarding performance in packets per second.*



(b) *Forwarding performance in Gigabits per second.*

Figure 5: *Forwarding performance for OVS without the microflow cache enabled, for input traffic of 60 Gbps.*

proves its forwarding performance, this feature relies on protocol-*dependent* features (specifically, that the packet has a five-tuple in the first place). Because our goal is to evaluate forwarding rates for protocol-independent switches, we disabled OVS's microflow cache so that we could compare PISCES, a protocol-independent switch, with a version of OVS that has no protocol-dependent optimizations. Comparing PISCES performance to that of OVS with microflow caching disabled thus offers a more "apples-to-apples" performance comparison.

Calibrating OVS to enable performance comparison. To allow us to more accurately measure the cost of parsing for both OVS and PISCES in subsequent experiments, we begin by establishing a baseline for Open vSwitch performance with minimal parsing functionality. To minimize the cost of parsing, we disabled the OVS parser, which ordinarily parses a comprehensive fixed set of headers, so that it reports only the input port. After this change, we sent test traffic through the switch with a trivial flow table that matches every packet that ingresses on port 1 and outputs it to port 2.

Figures 5a and 5b show the maximum throughput that our setup achieved with OVS, with and without the microflow cache, for 60-Gbps traffic. For 64-byte packets, disabling the microflow cache reduced performance by about 35%, because a lookup in the OVS megaflow cache consumes five times as many cycles as the microflow cache (Table 4). For small packets, the OVS switch is CPU-bound on lookups; thus, in

Switch Components	With MicroFlow	Without MicroFlow
Parser	19.0	18.9
MicroFlow Cache	18.9	—
MegaFlow Cache	—	92.2
Slow Path	—	—
Fast-Path Actions	39.9	38.8
End-to-End	100.6	166.0

Table 4: *Average number of cycles per packet consumed by each element in the virtual switch when processing a 64-byte packet.*

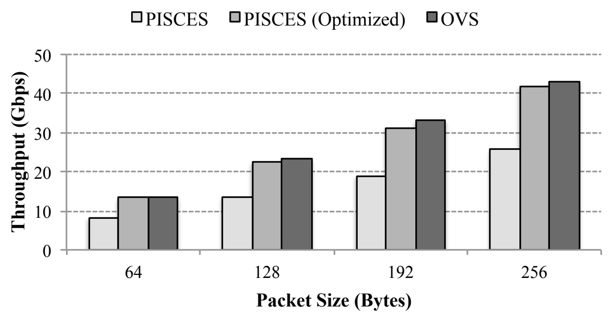


Figure 6: *Throughput comparison of L2L3-ACL benchmark application between OVS and PISCES.*

this operating regime, the benefit of the microflow cache is clear. With this calibration in mind, for the remainder of the evaluation section, we use the forwarding performance for OVS with the microflow cache disabled as the basis for our performance comparison to PISCES.

5.2.2 End-to-end performance

We next measured the forwarding performance of a real-world network application for both OVS and PISCES. This evaluation provides a clear illustration of the end-to-end performance costs of programmability. We selected a realistic and relatively complex application where both switch implementations provided all packet processing features to provide a fair performance comparison of PISCES in realistic network settings.

This application, which is shown in Figure 7 and which we call "L2L3-ACL", performs the following operations:

- Parse Ethernet, VLAN, IP, TCP and UDP protocols.
- Perform VLAN encapsulation and decapsulation.
- Perform control-flow and match-action operations according to Figure 7 to implement an access control list (ACL).
- Set Ethernet source, destination, type and VLAN fields.
- Decrement IP's TTL value.
- Update IP checksum.

Table 5 shows the forwarding performance results for this application. The most important rows are the last two, which show a "bottom line" comparison between OVS and PISCES,

Switch	Optimizations	Parser	MegaFlow Cache	Fast-Path Actions	End-to-End (Avg.)	Throughput (Mbps)
PISCES	Baseline	76.5	209.5	379.5	737.4	7590.7
	Inline Editing	-42.6	—	+7.5	-45.4	+281.0
	Inc. Checksum	—	—	-231.3	-234.5	+4685.3
	Action Specialization	—	—	-10.3	-9.2	+191.2
	Parser Specialization	-4.6	—	—	-7.6	+282.3
	Action Coalescing	—	—	-14.6	-14.8	+293.0
	All optimizations	29.7	209.0	147.6	425.8	13323.7
OVS	—	43.6	197.5	132.5	408.7	13497.5

Table 5: *Improvement in average number of cycles per packet, consumed by each element in the virtual switch when processing 64-byte packet, for L2L3-ACL benchmark application.*

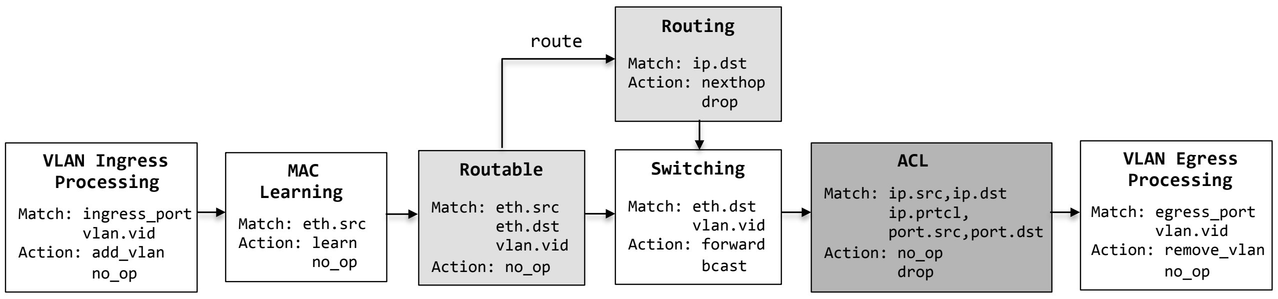


Figure 7: *Control flow of L2L3-ACL benchmark application.*

after we apply all compiler optimizations. These results show that both the average number of CPU cycles per packet and the average throughput for PISCES with all compiler optimizations is comparable to OVS with microflow caching disabled: both require just over an average of 400 CPU cycles per packet, and both achieve throughput of just over 13 Gbps—*a performance overhead of less than 2%*. Figure 6 demonstrates that this result also holds for larger packet sizes. In all cases, PISCES with compiler optimizations achieves performance comparable to OVS.

Next, we discuss in more detail the performance benefits that each compiler optimization achieves for this end-to-end application.

Benefits of individual compiler optimizations. P4 supports post-pipeline editing, so we started by compiling L2L3-ACL with post-pipeline editing. PISCES required an average of 737 cycles to process a 64-byte packet. Packet parsing and fast-path actions are primarily responsible for these additional CPU cycles. As our microbenchmarks demonstrate (Section 5.2.3), if the number of adjustments to packets are less than eight, using inline-editing mode provides better forwarding performance. Based on that insight, our compiler compiles PISCES with inline-editing, which reduces the number of cycles consumed by the parser by about 56%. However, fast-path actions cycles slightly increased (still 255 cycles more than OVS).

Next, we introduced incremental checksum updates to reduce the number of cycles consumed by the fast-path actions. The only IP field that is modified is TTL, but the full checksum verify and update design supported by P4 abstract model runs the checksum over entire headers once at the ingress and once at egress. For our P4 program, we specified that we wanted to use incremental checksum. Using this knowledge, instead of recalculating checksum on all header fields, using data flow analysis on the P4 program (MAT and control-flow), the P4 compiler determined that the pipeline modified only the TTL and adjusted the checksum using only that field, which reduced the number of cycles consumed by the fast-path actions by 59.7%, a significant improvement. However, PISCES was still consuming 23.24 more cycles than OVS.

To further improve the performance we applied action specialization and coalescing, and parser specialization. This brought the number of cycles consumed per packet by PISCES to 425.82.

Performance benefits of parser specialization. A protocol-independent switch only needs to parse the packet-header fields for the protocols defined by the programmer. The PISCES compiler can optimize the parser further to only parse the header fields that the switch needs to process the packet. To evaluate the potential benefits of this specialization, we repeated our end-to-end performance evaluation using two subsets of the L2L3-ACL program: the “L2L3” program, which does not perform the ACL functions, and the “L2” program, which manipulates the Ethernet and VLAN headers and performs VLAN encapsulation, but which does not parse any IP headers or decrement the TTL (and thus does not update the IP checksum). In terms of the control flow from the original “L2L3-ACL” benchmark program from Figure 7, the

Switch	Programs	Optimizations	Parser	MegaFlow Cache	Fast-Path Actions	End-to-End (Avg.)	Throughput (Mbps)
PISCES	L2L3	Optimized	22.9	188.4	130.5	392.3	14159.1
OVS	L2L3	—	43.6	176.0	131.8	388.3	14152.2
PISCES	L2	Optimized	19.7	148.2	90.9	305.7	18118.5
OVS	L2	—	43.6	155.2	78.7	312.1	17131.3

Table 6: *Improvement in average number of cycles per packet, consumed by each element in the virtual switch when processing 64-byte packet, for L2L3 and L2 benchmark applications.*

"L2L3" program removes the dark grey `ACL` tables, and the "L2" program additionally removes the light grey `Routable` and `Routing` tables.

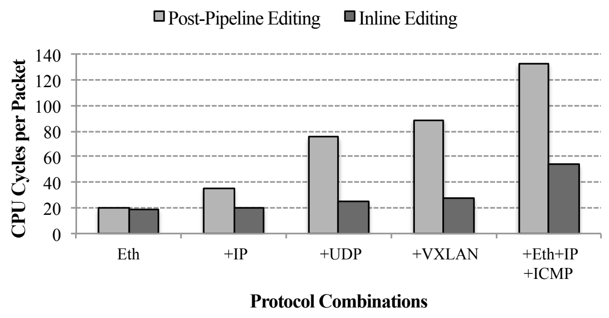
Table 6 compares the forwarding performance of OVS and PISCES for these two programs. For L2L3, PISCES consumed 4 more cycles per packet than OVS. However, note PISCES improvement in parsing performance: compared to L2L3-ACL, parsing in L2L3 is about 7 cycles per packet cheaper. OVS uses a fixed parser, so its cost remains constant. Parser specialization removed redundant parsing of fields from the parser that are not used in the control-flow (*i.e.*, TCP and UDP headers). Because OVS does not know the control-flow and MAT structure a priori, its parser cannot achieve the same specialization. In the case of the L2 application, the parser could specialize further, since it needed only to parse Ethernet headers. In this case, PISCES can actually process packets *more quickly* than the protocol-dependent switch.

5.2.3 Microbenchmarks

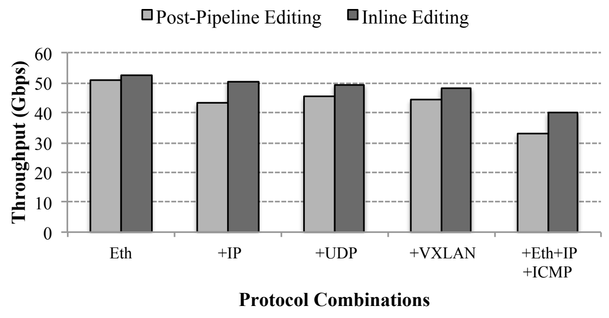
We now evaluate the performance of individual components of our PISCES prototype. We focus on the parser and actions, which are applied on every incoming packet and have the largest effect on performance. We now benchmark how increasing complexity in both parser and actions affect the overall performance of PISCES.

Parser performance. Figure 8a shows how per-packet cycle counts increase as the P4 program parses additional protocols, for both post- and inline-editing modes. To parse only the Ethernet header, the parser consumes about 20 cycles, in either mode. As we introduce new protocols, the cycle count increases, more rapidly for post-pipeline editing, for which the switch creates an extra copy of the protocol headers for fast-path actions. For the largest protocol combination in Figure 8a, the parser requires about 133 cycles (almost 6× Ethernet alone) for post-pipeline editing and 54 cycles for inline-editing. Figure 8b shows how the throughput decreases with the addition of each new protocol in the parser. For 64-byte packets at 60 Gbps, switching throughput decreases about 35%, from 51.1 Gbps to 33.2 Gbps, for post-pipeline editing and about 24%, from 52.4 Gbps to 40.0 Gbps, for inline editing.

Fast-path action performance. Performance-wise, the dominant action in a virtual switch is the set-field (or modify-field) action or, in other words, a write action. Figure 9 shows the



(a) *CPU cycles.*



(b) *End-to-end throughput.*

Figure 8: *Effect on parser CPU cycles and end-to-end throughput as more protocols are added to the parser.*

per-packet cost, in cycles, as we increase the number of set-field actions in the fast path for both post- and inline-editing modes. In post-editing mode, we apply our changes to a copy of the header fields (extracted from the packet) and at the end of the pipeline execute a "deparse" action that writes the changes back to the packet. The "deparse" bar shows how deparsing consumes about 99 cycles even if no fields are modified, whereas inline editing has no cost in this case. As the number of writes increases, the performance difference between the two modes narrows. For 16 writes, this difference is 20 cycles less than for a single write. Still, in both cases, the number of cycles increases. For post-editing case, 16 writes consumes 354 cycles, about 3.6× a single write; for inline editing, 16 writes consumes 319 cycles, or about 5.6×.

We also measured cycles-per-packet for adding or removing headers. Figures 10 and 11 show cycles-per-packet for an

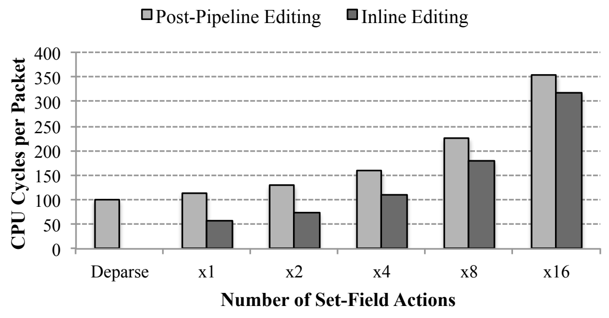


Figure 9: *Fast Path Set-Field Action Performance.*

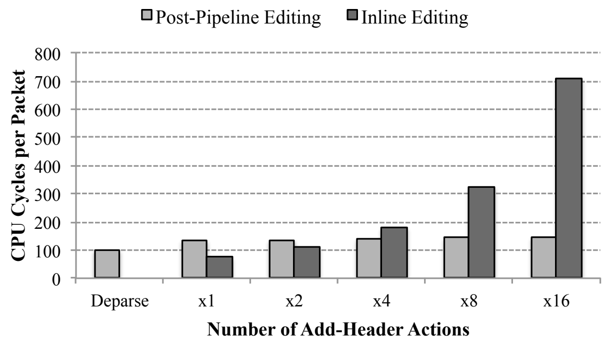


Figure 10: *Fast Path Add-Header Performance.*

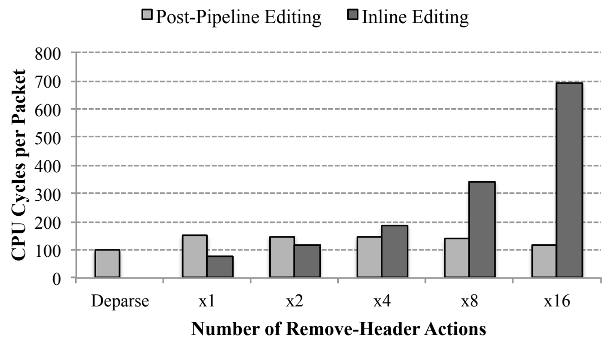


Figure 11: *Fast Path Remove-Header Performance.*

increasing number of add-header and remove-header actions, respectively, in the post- and inline-editing modes.

For the `add_header()` action, for inline-editing mode, the number of cycles doubles for every new action. This is because these actions are applied directly on the packet, adjusting the packet size each time, whereas post-editing adjusts the packet size only once, in the "deparse" action, so that the number of cycles consumed remains almost constant. For a single `add_header()` action, post-editing cost is higher, but for four or more actions the inline-editing mode is more costly. For 16 add-header actions, inline editing consumes 577 more cycles per packet than post-editing.

We observe a similar trend for `remove_header()`. There is one additional wrinkle: as the number of `remove_header()` actions increase, the cost of post-pipeline editing actually decreases slightly, because fewer bytes need to be adjusted in the packet as the packet shrinks. As we increase the number of remove-header actions from 1 to 16, the per-packet cycle count decreases by about 21%. This led us to the following rule of thumb: for fewer than 8 packet-size adjustments (*i.e.*, add- and remove-header actions), the compiler uses inline-editing; otherwise, it applies post-pipeline editing, as the added number of cycles required by the parser to generate a copy of the parsed packet headers is offset by the number of cycles required by the add/remove header actions in the inline-editing mode.

Slow-path forwarding performance. When OVS must send all packets to the slow path, it takes on average about 3,500 cycles to process a single packet (about 50× a microflow cache hit). In this case, the maximum packet forwarding rate is about 0.66 Mpps regardless of packet size. This per-packet cycle count for slow-path processing was for the simplest possible program that sends every packet to the same output port. Most real packet processing program would require significantly more cycles. For example, for the L2L3-ACL program, slow-path processing required anywhere from 30,000 to 60,000 CPU cycles per packet. These performance numbers indicate the importance of the megaflow cache optimizations that we described in Section 4.3 to reduce the number of trips to the slow path. Clearly, the number of trips to the slow path depends on the actual traffic mix (because this affects cache hit rates in the megaflow cache), so it is difficult to state general results about the benefits of these optimizations, but computing the slowdown as a result of cache misses is straightforward.

Control flow. Control flow in Open vSwitch, and thus in PISCES, is implemented in the slow path. It has a small one-time cost, which is impossible to separate from slow path performance in general, at the setup of every new flow.

6 Related Work

PISCES is a software switch whose protocols and packet-processing functions can be specified using a high-level domain-specific language for packet processing. Although PISCES uses P4 as its high-level language and OVS as its software switch, previous work has developed both domain specific languages for packet processing and virtual software switches, where our approaches for achieving protocol independence and efficient compilation from a DSL to a software switch may also apply.

Domain specific languages for packet processing. The P4 language provided the main framework for protocol independence [8]; PISCES realizes protocol independence in a real software switch. P4 itself borrows concepts from prior work [5, 18, 31]; as such, it may be possible to apply similar concepts that we have implemented in PISCES to other high-level languages. Although our current PISCES prototype compiles P4 to OVS source code, the concepts and optimizations that we have developed could apply to other high-level langauges and target switches; an intermediate representation such as NetASM [44] could ultimately provide a mechanism for a compiler to apply optimizations for a variety of languages and targets. Languages such as Pyretic [42] and Frenetic [19] are domain-specific languages that specify how packets should be processed by a fixed-function OpenFlow switch, not by a protocol-independent switch.

Virtual software switches. Existing methods and frameworks for building software switches like Linux Kernel [30], DPDK [24], Netmap [43], and Click [29] require intimate knowledge about the underlying implementation and, thus, make it difficult for a network programmer to rapidly adapt and add new features to these virtual switches. PISCES, on the other hand, allows programmer to specify packet processing behavior independent of the underlying implementation details. Open vSwitch [39] provides interfaces for populating its match-action tables but does not provide mechanisms to customize protocols and actions.

Other programmable switches. Software routers such as RouteBricks [14], PacketShader [22], and GSwitch [47] rely on general-purpose processors or GPUs to process packets; these designs generally focus on optimizing server, network interface, and processor scheduling to improve the performance of the software switch. These switches do not enable programmability through a high-level domain-specific language such as P4, and they also do not function as hypervisor switches. CuckooSwitch [48] can be used as a hypervisor switch. However, it focuses on providing fast FIB lookups by using highly-concurrent hash tables based on Cuckoo hashing [36], and, also does not provide a high-level domain-specific language to configure the switch. SwitchBlade [4] enables some amount of protocol customization and forwards packets at hardware speeds, but also acts as a standalone switch and requires an FPGA as a target.

Measuring performance. Previous work has both measured [7, 16] and improved [11, 12, 38, 39] the performance of software virtual switches. Work on measurement has converged on a set of performance metrics to compare various switch architectures and implementations; our evaluation uses these metrics to compare the performance of PISCES to that of other virtual switches.

Measuring complexity. A number of metrics for measuring the complexity and maintainability of a program written in a DSL are developed in software engineering [10, 23, 25, 26, 33]. One of the goal of PISCES is to make it easier for the programmer to develop and maintain code. For our evaluations, we have taken these metrics from software engineering to to evaluate the complexity of writing a program in P4 vs. directly modifying the OVS source code in C.

7 Conclusion

The increasing use of software hypervisor switches in virtualized data centers has introduced the need to rapidly modify the packet forwarding behavior of these software switches. Today, modifying these switches requires both intimate knowledge of the switch codebase *and* extensive expertise in network protocol design, making the bar for customizing these software switches prohibitively high. As an alternative to this mode of operation, we developed PISCES, a programmable, protocol-independent software switch that allows a protocol designer to specify a software switch's custom packet processing behavior in a high-level domain-specific language (in our case, P4); a compiler then produces source code for the underlying target software switch (in our case, OVS). PISCES programs are about 40 times more concise than the equivalent programs in native code for the software switch. We demonstrated that, with appropriate compiler optimizations, this drastic reduction in complexity comes with hardly any performance cost compared to the native software switch implementation.

Our prototype demonstrates the feasibility of a protocol-independent software switch using P4 as the programming language and OVS as the target switch. Moreover, our techniques for software switch protocol independence and for compiling a domain-specific packet-processing language to an efficient low-level implementation should generalize to other languages and targets. One way to achieve language and target-independence would be to first compile the domain-specific languages to a protocol-independent high-level intermediate representation (HLIR) such as protocol-oblivious forwarding [45] or NetASM [44], then apply the techniques and optimizations from PISCES to the HLIR.

Another future enhancement for PISCES is to enable custom parse, match, and action code to be dynamically loaded into a running protocol-independent switch. Our current PISCES prototype requires recompilation of the switch source code every time the programmer changes the P4 specification. In certain instances, such as adding new features and protocols to running production switches or temporarily altering protocol behavior to add visibility or defend against an attack, dynamically loading code in a running switch would be valuable. We expect future programmable protocol-independent software switches to support dynamically loading new or modified packet-processing code.

It is too early to see the effects of PISCES on protocol development, but the resulting code simplicity should make it easier to deploy, implement, and maintain custom software switches. In particular, protocol designers can maintain their custom software switch implementations in terms of a high-level domain-specific language like P4 without needing to track the evolution of the (larger and more complex) underlying software switch codebase. The ability to develop proprietary customizations without having to modify (and track) the source code for a software switch such as OVS might also be a selling point for protocol designers. We intend to study and characterize these effects as we release PISCES and interact with the protocol designers who use it.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques*. Addison Wesley, 1986.

[2] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese, et al. CONGA: Distributed Congestion-aware Load Balancing for Datacenters. In *ACM SIGCOMM*, pages 503–514, Chicago, IL, Aug. 2014.

[3] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *ACM SIGCOMM*, Seattle, WA, Aug. 2008.

[4] B. Anwer, M. Motiwala, M. bin Tariq, and N. Feamster. SwitchBlade: A Platform for Rapid Deployment of Network Protocols on Programmable Hardware. New Delhi, India, Aug. 2010.

[5] G. Back. DataScript: A specification and scripting language for binary data. In *Generative Programming and Component Engineering*, pages 66–77. Springer, 2002.

[6] W. Bai, L. Chen, K. Chen, D. Han, C. Tian, and H. Wang. Information-agnostic flow scheduling for commodity data centers. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Oakland, CA, May 2015.

[7] A. Bianco, R. Birke, L. Giraudo, and M. Palacin. OpenFlow switching: Data plane performance. In *IEEE International Conference on Communications*, pages 1–5. IEEE, 2010.

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlessinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communications Review*, July 2014.

[9] 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*, pages 99–110. ACM, Aug. 2013.

[10] D. Coleman, D. Ash, B. Lowther, and P. Oman. Using metrics to evaluate software system maintainability. *Computer*, 27(8):44–49, 1994.

[11] J. Corbet. BPF: The universal in-kernel virtual machine. `http://lwn.net/Articles/599755/`, 2014.

[12] J. Corbet. Extending extended BPF. `http://lwn.net/Articles/603983/`, 2014.

[13] M. Dillon and T. Winters. Network functions virtualization in home networks. Technical report, Open Networking Foundation, 2015. `https://goo.gl/alor91`.

[14] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting parallelism to scale software routers. In *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[15] N. Dukkipati, G. Gibb, N. McKeown, and J. Zhu. Building an RCP (Rate Control Protocol) test network. In *Hot Interconnects*, volume 15, 2007.

[16] P. Emmerich, D. Raumer, F. Wohlfart, and G. Carle. Performance characteristics of virtual switching. In *IEEE International Conference on Cloud Networking*, pages 120–125. IEEE, 2014.

[17] P. Emmerich, F. Wohlfart, D. Raumer, and G. Carle. MoonGen: A scriptable high-speed packet generator. *arXiv preprint arXiv:1410.3322*, 2014.

[18] K. Fisher and R. Gruber. PADS: A domain-specific language for processing ad hoc data. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 295–304. ACM, June 2005.

[19] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *International Conference on Functional Programming*, Sept. 2011.

[20] T. M. Gil and M. Poletto. Multops: a data-structure for bandwidth attack detection. In *USENIX Security Symposium*, pages 23–38, Washington, DC, 2001.

[21] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: a scalable and flexible data center network. In *ACM SIGCOMM*, Barcelona, Spain, Aug. 2009.

[22] S. Han, K. Jang, K. Park, and S. Moon. Packetshader: a gpu-accelerated software router. *ACM SIGCOMM*, Aug. 2010.

[23] N. Heirbaut. Two implementation techniques for Domain Specific Languages compared: OMeta/JS vs. JavaScript. `http://homepages.cwi.nl/~storm/theses/heirbaut2009.pdf`, Oct. 2009.

[24] Intel. DPDK: Data Plane Development Kit. `http://dpdk.org/`, 2013.

[25] S. H. Kan. *Metrics and models in software quality engineering*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[26] C. Kaner et al. Software engineering metrics: What do they measure and how do we know? In *In METRICS 2004. IEEE CS*. Citeseer, 2004.

[27] D. Katabi, M. Handley, and C. Rohrs. Congestion control for high bandwidth-delay product networks. *ACM SIGCOMM*, Aug. 2002.

[28] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker. In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*, London, England, Aug. 2015. Demo Session.

[29] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[30] Linux kernel archives. `http://kernel.org/`, 1997.

[31] P. J. McCann and S. Chandra. Packet types: Abstract specification of network protocol messages. In *ACM SIGCOMM*, pages 321–333. ACM, Aug./Sep. 2000.

[32] Network Working Group. RFC 1624: Computation of the Internet checksum via incremental update, May 1994.

[33] P. Oman and J. Hagemeister. Metrics for assessing a software system's maintainability. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 337–344. IEEE, 1992.

[34] OpenFlow. `http://www.opennetworking.org/sdn-resources/technical-library/`, January 2015.

[35] Open vSwitch Git repository. `https://github.com/openvswitch/ovs`, October 2015.

[36] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[37] V. Parulkar. Programming the network dataplane in P4. `http://goo.gl/6FAzql`, Nov. 2015.

[38] B. Pfaff. P4 parsing in Open vSwitch, June 2015. Talk at the P4 workshop, `http://schd.ws/hosted_files/p4workshop2015/64/BenP-P4-Workshop-June-04-2015.pdf`.

[39] B. Pfaff, J. Pettit, T. Koponen, E. 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 *Networked Systems Design and Implementation*, May 2015.

[40] S. Previdi et al. *SPRING Problem Statement and Requirements*. IETF, June 2015. `https://datatracker.ietf.org/doc/draft-ietf-spring-problem-statement/`.

[41] Proxmox Virtual Environment. `https://www.proxmox.com/en/proxmox-ve`.

[42] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker. Modular SDN programming with Pyretic. *USENIX; login*, 38(5):128–134, 2013.

[43] L. Rizzo. netmap: A novel framework for fast packet I/O. In *USENIX Annual Technical Conference*, pages 101–112, 2012.

[44] M. Shahbaz and N. Feamster. The case for an intermediate representation for programmable data planes. In *Proceedings of the ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*. ACM, June 2015.

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

[46] V. Srinivasan, S. Suri, and G. Varghese. Packet classification using tuple space search. In *ACM SIGCOMM*, 1999.

[47] M. Varvello, R. Laufer, F. Zhang, and T. Lakshman. Multilayer packet classification with graphics processing units. In *ACM International on Conference on Emerging Networking Experiments and Technologies*, Dec. 2014.

[48] D. Zhou, B. Fan, H. Lim, M. Kaminsky, and D. G. Andersen. Scalable, high performance Ethernet forwarding with CuckooSwitch. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 97–108. ACM, 2013.