NOSIX: A Lightweight Portability Layer for the SDN OS

Minlan Yu
USC

Andreas Wundsam
Big Switch Networks

Muruganantham Raju[*]
USC

ABSTRACT

In spite of the standardization of the OpenFlow API, it is very difficult to write an SDN controller application that is portable (i.e., guarantees correct packet processing over a wide range of switches) and achieves good performance (i.e., fully leverages switch capabilities). This is because the switch landscape is fundamentally diverse in performance, feature set and supported APIs. We propose to address this challenge via a lightweight portability layer that acts as a rendezvous point between the requirements of controller application and the vendor knowledge of switch implementations. Above, applications specify rules in virtual flow tables annotated with semantic intents and expectations. Below, vendor specific drivers map them to optimized switch-specific rule sets. NOSIX represents a first step towards achieving both portability and good performance across a diverse set of switches.

Categories and Subject Descriptors

C.2.1 [**Computer-communication networks**]: Network Architecture and Design; C.4 [**Performance of Systems**]: Design studies

Keywords

Software-Defined Networks, OpenFlow, Switch Diversity, Portability

1. INTRODUCTION

A core promise of SDN is that SDN users are freed from being locked to specific hardware vendors through the use of standardized APIs—they should be able to "mix and match" different types and vendors of switches in their network, without having to change the controller application. However, in spite of the standardization of the OpenFlow API, it is very difficult today to write an SDN controller application that is truly *portable*, i.e., that guarantees *correct* packet processing and has *good performance* over a wide range of switches.

The reason for this challenge is that the switch landscape is fundamentally diverse: Switches are built in software or hardware, and optimized for different trade-offs of cost and scale. Software switches [3, 10] profit from the fast CPUs and I/O in modern servers on the control plane. On the data plane however, their performance varies with the load on the server and the characteristics of the installed flows. In contrast, hardware switches [5, 7] only carry a relatively weak CPU, and impose strict resource constraints. Once a flow is installed, however, they provide full bisection data plane bandwidth. Among them, there is significant variety in the number of flow tables, the number of flows that can be installed in each table, and the matches and actions supported. Pipelines are fixed or partially flexible. Table sizes can vary with their contents. While recent work [11] makes the case that switch pipelines *should* be fully flexible, in today's reality they are not. We believe such switch diversity is not a short term phenomenon. Consequently, the SDN community should address the portability problem.

Fundamentally, the challenge of portability is thus that expectations of the application have to be matched to the feature-set and performance characteristics of the switch that forwards the packets. To bridge this gap, there are two basic approaches:

Top down: Generic applications across all the switches. In a top-down approach, operators write applications using a generic representation, independent of the actual pipeline and capabilities of the switch and without any specifications of device-level properties. For example, OpenFlow 1.0 only uses the TCAM table of a switch, which makes it easy to write portable programs, but leaves most of a typical switch pipeline unused. Other top-down approaches either represent the network as a single virtual switch with unconstrained resources [20], or provide a declarative flow programming language that applications use [16]. A compiler then translates the higher layer application commands to optimized OpenFlow messages. This approach enables portability and simplifies the programming of applications, but the compiler is highly complex to design and necessarily incurs overhead. Without the help of applications, it is difficult to come up with optimal solutions, as flow characteristics are not known in advance. Also, a high level abstraction is less appropriate for applications "close to the metal" that require access to low-level details on the wire. It is also challenging to rewrite a switch-specific compiler for each high-level language.

Bottom up: Switches expose capabilities to the applications or the runtime system. Recent OpenFlow specifications have shifted towards a bottom-up approach, where switches expose a more detailed model of their internal architecture. However, it is difficult to express the entire forwarding behavior in a generic language. OpenFlow 1.3/1.4 exposes multiple switch tables, but the actual pipeline logic, and individual contracts and resource limits for these tables are still not included (e.g., TCAM may have flexible number of entries as discussed in Section 2). If the switch exposes too little information, applications are unable to fully optimize their performance. However, if the switch exposes too many details, application programmers may either simply not care about these details, or have to make choices without full confidence.

Neither of the above approaches is able to solve the diversity problem in the general case. Solving the problem of portability in SDN requires bringing together knowledge from both the application programmer (about what the application is semantically try-

[*]Authors in reverse alphabetical order.

ing to achieve, what its priorities are, what trade-offs it can make) and the switch vendor (about the exact feature set and performance characteristics of the switch).

As a new point in the design space, we propose a lightweight compatibility layer between the controller applications and the switches, called NOSIX. The idea is to combine both the application programmers' expertise of their requirements and the vendors' expertise of their switch architecture. In a NOSIX system, application writers group rules and annotate the groups with expectations and optimization goals, while vendors provide software device drivers that map these groups to the hardware. To retain the efficiency and richness of the OpenFlow programming model, NOSIX does not fully isolate the application from the switch characteristics. Instead, it gives applications a choice: They can omit details they do not care about, or provide specific requirements where they do. This enables more efficient resource usage in the switches, fosters switch innovations, while simplifying controller applications.

In presenting NOSIX, we hope to widen the discussion in the field to include approaches that are practical with currently available hardware, where switch pipelines are only partially flexible and not all tables provide a TCAM-like feature-set. The concrete API is intended to serve as an example that clarifies and illustrates this approach.

2. SWITCH DIVERSITY

Today's switch implementations differ substantially in both the data plane and control plane. According to Open Networking Foundation [8], there are 25 OpenFlow hardware and software switch products from 15 companies. They differ in both the data plane and the controller-switch interactions [28, 18].

2.1 The pipeline of flow tables

Today's OpenFlow specifications, even version 1.4, only expose part of "the switch brain". To illustrate the complexity, we discuss a model of a traditional ASIC-based hardware switch based on our hands-on experiences [4]. The switch has a partially flexible pipeline of tables that can each match on specific attributes and perform, which includes: (1) the classification table at each input port, which classifies packets based on input port and VLAN tag id. This table has about 1000 entries and sets the metadata to be used for the following tables. (2) the L2 table, which matches on the tuple of exact destination MAC address and VLAN ID. The table has hundreds of thousands of entries, and can send the packets either directly out (skipping the following tables) or further through the pipeline. (3) the TCAM-based ACL table, which matches on arbitrary fields and has a wide range of possible actions. The table has limited size of only a few thousands of entries, but the size is not constant: the table is subdivided into *slices*, which need to be configured based on the number of bits that need to be matched. For instance, when we match on IPv6 addresses, a slice has to be reconfigured to "double wide" mode, reducing the number available table entries. Given the complexity of switch design details, it is hard to expose all the features to the applications. (4) the SRAM-based L3 table, which matches on IPv4 addresses using longest prefix match. This table has tens of thousands of entries and can rewrite the destination IP addresses.

The details of the table pipelines vary across switches [18]. Switches may be able to skip pipeline stages or loop back to earlier stages. They may be able to output a packet twice in the pipeline, but only modify it once. In multi-module switches, TCAMs can be at a central location and managed by a single entity (e.g., Cisco Catalyst 4500 Supervisor) or they can exist on each line card and managed by local CPUs (e.g., HP ProCurve 5400). OVS [10], a software switch, has a kernel-level hash table that handles micro flow rules, and a user-space table that handles wildcard rules and transparently inserts micro flow rules into the kernel-level hash table.

Switch tables differ in their sizes, supported matches and actions. For OF 1.0, the HP ProCurve 5406zl supported 1500 flows, while the NEC PF5820 supported 750 flows [25]. As discussed before, the size may even vary with content. For matches, there exist multiple types of TCAMs optimized on specific matches in order to serve different goals (e.g., FIB, ACL, QoS) and performance [12]. For example, one of three hardware switches evaluated by Rotsos [28] did not support IP address rewriting. The Intel switch [6] supports more complex and flexible flow match specifications and actions. OVS supports the full set of matching and actions and a conceptually unbounded number of rules.

Switches have different packet processing performance in the data plane. Hardware switches can typically forward packets at full line-rate, independent of the switch CPU and host bus. Features not directly supported by the hardware may be implemented in software on the *slow path*, which is orders of magnitude slower. In contrast, software switches may expose different performance for rules kept in the kernel or in the user space [24], and may deteriorate with growing flow table size [28, 15].

Implications: This diversity creates the challenge of if and where (in which flow-table) an application should install its rules. It is challenging for applications to fully understand and leverage the features of different switches for the best performance. It is also challenging for operators to write generic applications while achieving good performance.

2.2 Controller-switch interactions

The controller-switch interactions include rule updates from the controller to the switch and the state updates from the switch to the controller. In hardware switches, rule and state updates may incur different delays due to limits of the switch CPU, or scheduling conflicts to the underlying ASIC[28]. Software switches can update rules and answer state queries faster but may still be delayed under heavy CPU load. Depending on their architecture, switches may be able to perform groups of changes atomically or not.

Applications often keep a local copy of the flow table at the controller and *synchronize* the local flow tables with those at the switch. However, this is non-trivial because switches may automatically expire some rules and may delay table modification commands. When the control channel is temporarily interrupted and recovered, the application no longer knows the state of the switch. The most common practice is to clear the switch flow table in this case, and repopulate it from scratch. While this solution restores to the flow table to a known state, it also breaks forwarding during recovery. Applications with a more sophisticated strategy may query the switch flow table on reconnect, and only delete surplus and add rules in the switch flow table. This does not break the forwarding, but requires the entire flow table to be transmitted from the switch to the controller. Also, rules that have been correctly expired by the switch due to timeouts may be re-installed by the controller.

Implications: Communication between the controller and the switches incurs delays and limited bandwidth. As a result, it is challenging for applications to understand the performance of such interactions for different switches, and optimize their performance accordingly.

2.3 Diversity is fundamental

Some feature and performance differences among switches stem from the fact that OpenFlow and SDN are still nascent. First generation SDN switches were built on top of existing switch fabrics, limiting the achievable feature set. While new proposals advocate more flexible pipelines [11], these are not yet widely available. At this point, all parts of the SDN stack are still evolving rapidly (e.g., NoviFlow [1] builds OpenFlow switches on NPUs), resulting in inconsistent feature sets and support for new features (e.g., *Groups* in OpenFlow 1.1+). We expect that this evolution-based diversity will persist for several years until the SDN ecosystem stabilizes.

However, even in the long term, switches will continue to be diverse due to economic pressures. SDN protocols are not meant to standardize switch implementations but only the controller APIs. There will always be switch implementations with different trade-offs of cost and scale. Switches will be customized for different applications and markets. For example, we need different types of switches for wide area networks and data center networks, at the edge and in the core of networks. Both hardware and software will continue to evolve independently. For example, when new SDN switches support larger TCAMs and more tables, applications written for OpenFlow 1.0 need to be rewritten to leverage these new features.

Implications: Switch diversity not only exists in today's SDN, but will remain in the future because of diverse markets and hardware evolution. As a result, applications always face the trade-off of portability (i.e., running on all types of switches) and performance (i.e., optimizing for individual switches).

3. THE NOSIX APPROACH

We now discus a hybrid approach that attempts to strike a balance between portability and performance across switches. The key idea is to introduce an abstraction that separates the application's expectations from the switch-specific implementations. We argue that the applications should choose a pipeline model that fits their goal with some freedom. They then specify their *requirements* about rules, updates, and notifications in annotations. It is then switch vendors' job to choose the best way to implement these expectations by leveraging the knowledge of their own switches.

Note that our intention is to discuss the approach rather than propose a fixed API, so we focus on highlighting key design points: (1) *A pipeline of virtual flow tables* to represent application expectations on what rules should be processed at the switch. Each virtual flow table contains flow-based rules with priorities that match packets on different header fields, take actions, and accumulate counters. The tables are defined by the application and assembled in a custom pipeline. (2) A *switch-specific driver* that maps the VFT pipeline onto the physical pipeline available on the switch. The driver performs a *static mapping* based on the information in the virtual pipeline, and its annotations, and its knowledge of the properties of the physical pipeline. This driver can be placed either at a low layer in the controller OS, or on the switch itself.

3.1 Virtual flow tables (VFTs)

Virtual Flow Tables (VFTs) are the basic components in NOSIX. The applications can freely define the rules without worrying about either the resource constraints and feature sets in the forwarding plane, or the delay and throughput of updates and notifications. We first give an example to show how these virtual flow tables work (Figure 1). Assume a virtual networking application, which distributes traffic to different virtual networks. The application can specify a pipeline of VFTs: First, a VFT that classifies which virtual network the traffic belongs to (classify_host) based on the input port and/or the VLAN tag, and sets some metadata identifying the virtual network the host is on. Then, in each virtual network, the application handles traffic differently based on the dl_proto field. For instance, the ARP rule table maps known MAC address to IP addresses. The ACL table performs access control of IP traffic and the L3 routing table decides the routing for the traffic. The key properties of our proposed virtual flow tables are summarized as follows:

VFTs are predefined by applications: Applications pre-define a pipeline of *virtual flow tables (VFTs)* and then install rules in these VFTs as they operate. We only allow applications to pre-define the pipeline because physical switch pipeline is hard for dynamic reconfiguration. In the general case, rules cannot be reshuffled without intermittent data plane impact. While current switch TCAMs can be *sliced* and *reconfigured* to adapt to different application scenarios (e.g., enabling or disabling matches on Ethernet, IPv4 or IPv6 addresses to optimize space), these operations typically cannot be performed online. Note that while the *structure* of the pipeline is static, the rules themselves are not. Applications can still dynamically insert rules into the VFTs on-line at any time.

VFTs are not constrained by physical switch tables: The rules in the VFTs do not need to be always in the physical flow tables. For instance, in the example above, the application uses a dedicated table to handle the responses to ARP queries in the network. In the absence of specialized hardware support, the switch driver can emulate this table in software. NOSIX makes this decision based on the performance requirements of the application as expressed by the annotation. We can map a VFT to rules in multiple physical tables or map multiple VFTs into the same physical table. In the example in Figure 1, NOSIX may decide to *combine* the sequential virtual tables (classify_protocol and ACL) and install the combined rules from both tables into the same TCAM table.

Applications specify their expectations using annotations to VFTs: To enable NOSIX to make smart decisions in mapping VFTs, applications need to expose the properties of these VFTs and their expectations using annotations. In the following, we describe two examples of such annotations. This is not intended to be an exhaustive list. We expect the language of such annotations to be dynamic and grow as applications and switches develop and common patterns emerge.

Requirements: Applications can describe their *requirements* on how switches should handle the rules in each VFT, in order to guarantee good performance of their rule processing at switches. The requirements include which switch table to install the rules (e.g., a table optimized for matching on MAC and a table for longest prefix match), the rule processing rate (e.g., at least 500 rules per second), consistency requirements (e.g., per-flow consistency [27]).

These requirements represent mandatory preconditions that the application requires to perform according to its SLA. If the feature-set of the switch does not permit the requirements to be fulfilled, the application fails to run and the operator is notified to either upgrade the switch or change the requirements.

Promises: To increase the degrees of freedom of placing the rules, applications can give *promises* about their expected usage of the rules in each VFT. The example promises include flow size and rule update rate. For instance, ARP and DNS flows are often short lived and have only a few packets. Therefore, the VPT for ARP and DNS may have the promise of the flow size < 10 pkts/flow and the rule update rate < 5 flows/sec. By contrast, rules stemming from the migration of virtual machines are likely to match a large amount of packets at high rates, but these packets occur only sporadically. The application can install these rules in another VFT but with a promise of larger flow sizes but a lower rule update rate. With these promises, NOSIX can map a VFT to the right physical flow

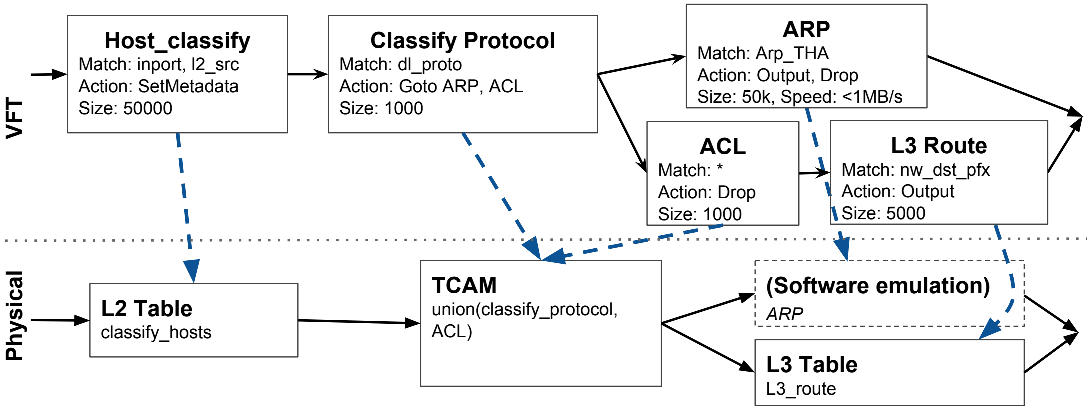

Figure 1: Example of virtual flow tables

tables and reserve more table entries for the other VFTs.

Figure 1 shows how an application annotates each table based on its expectations. For example, the application annotates the maximum size of each VFT, and the ARP VFT with a promise that ARP traffic would not go beyond 1Mbps. Note that we only discuss a few representative requirements and promises above. NOSIX can be extended to support more properties based on applications' requirements because they are not constrained by the switch implementations. For future work, we will study how to incentivize applications to tell the right requirements and promises.

Update rules in VFTs with transactions: To simplify the programming model for the application, NOSIX exposes a transaction-based API for applications to change rules in VFTs. There are two primitives: *start_transaction(consistency_level)* and *commit(wait)*. All the rule operations the applications have on the virtual flow tables after *start_transaction* must remain inactive until the *commit* command. The end state of the transaction is that either *all* or none of the operations are applied. The application can specify the *consistency_level* explicitly (e.g., no consistency, per-packet consistency, and per-flow consistency). The default consistency level is no consistency, which does not provide any guarantee that the changes are applied atomically on the switch.

The transaction language in NOSIX is defined on a per-switch basis. Network-wide semantics can be implemented on top of it [27, 13]. Note that NOSIX itself makes no guarantees or assumptions about the availability of transactions on the switch API. Instead, it provides a language for the application to specify that it requires such behavior. This enables varying implementations in the drivers, based on the options in the hardware. For instance, a driver may leverage their knowledge of operation orderings and may take advantage of *disabled table entries* in the hardware tables, where such features are available.

VFTs are defined based on rule properties instead of rule functions: Applications use VFTs to describe the properties of their rules but not their functions. Rules for the same function may be distributed on multiple VFTs. For example, an application may divide its access control rules into two VFTs, one with frequently changing rules for operators to isolate immediate threats, and the other with stable rules. In this way, NOSIX can map the two VFTs to different physical flow tables at switches. Rules with different functions may fit in the same VFT. For instance, if the access control rules and routing rules have similar promises and requirements (e.g., in terms of packet processing time and rule change rates), the application can install both types of rules in a single VFT and NOSIX does not need to distinguish them in the physical flow tables.

3.2 Switch drivers

Efficiently mapping the VFTs onto the physical switches requires detailed knowledge about the feature set and resource constraints of the switch. Despite the progress of OpenFlow specifications, describing these trade-offs in precision using a vendor-agnostic language is still difficult. For instance, switches may be able to partition their TCAMs, using different matching modes on different parts. The hash-based tables may experience different hash collisions, depending on the hash functions used. We argue that instead of striving to expose the entire complexity of the pipeline to application, the application should just expose its expectations via VFTs, and then switch drivers should map VFTs to physical switches (e.g., Figure 1). The switch drivers have the following key features:

Switch drivers are programmed by vendors: Vendors are in the best position to implement switch drivers because they understand their switches well. Switch drivers hide switch heterogeneity from applications and enable diversity and innovations of both the switch data plane and control plane. In the data plane, instead of following the same specification, vendors now have the flexibility to design their own features (e.g., the types and number of flow tables) and optimize its resource usage (the size and matching fields for each flow table), as long as they can use the driver to manage the rules based on applications' expectations. For example, the application may define one table for measurement rules and another for data rules, and the switch driver may merge and map them to the single TCAM table in a hardware switch. In the control plane, vendors can design different mechanisms to process rule updates and send network events and flow counters to the applications.

Switch drivers can be on the switch or on the controller: We do not make a strong case here about *where* the switch driver should be placed. The switch driver may reside on the switch itself. An advantage is that this approach maintains the clean separation between SDN programmer responsibility (controller) and switch vendor responsibility (on the switch). The disadvantage is that the need for standardization at the controller-switch boundary limits the rate of innovation possible. Note that today some vendors already use firmware to mimic OpenFlow features they cannot support in their hardware (e.g., packet encapsulation).

In contrast, the switch driver may also reside at a low level of the controller software stack, by moving part of the *switch control plane* to the controller (i.e., closer to the applications). There are several advantages: (1) The control channel between the controller and the switch may have delay churns and failures, leading to high complexity in programming applications. With the switch drivers, the vendors fully control the messages between the controller and the switch (e.g., update rules, send notifications), and thus can introduce different control protocols.[1] (2) Today, the limited CPU at some switches significantly limits the control plane performance and the switch feature set. With switch drivers, vendors can leverage the more powerful computing resources shared across switches in the controller, and decide whether to implement a feature in the data plane, the firmware, or the switch driver. (3) A single driver in the controller can manage all the switches belonging to the same vendor, enabling more switch-specific optimization. (4) It is much easier to program the software-based switch driver at the controller than at the switch because it is easy for debugging and reuse modules. That said, some components of the switch drivers should be located at the switches for lower latency (e.g., rule rearrangement in the TCAM). If the driver is directly embedded into the controller software, it has to be rewritten for each individual controller. This can be circumvented by having a language-agnostic API inside the controller OS, e.g., through a virtual file system API [23].

Switch drivers automatically synchronize between VFTs and physical tables: When an application modifies the rules in the virtual flow table, the switch drivers should update the rules in the switches accordingly based on the application's requirements. The switch drivers also handle flow expiration at switches, and re-synchronize the physical tables with VFTs when the controller-switch channel fails.

Switch drivers also synchronizes the flow-level counters from the physical flow tables at switches to the virtual flow tables. These drivers optimize the rule update and counter collection performance based on the switch feature set and the trade-off of features and performance. For example, if the switch supports more advanced ways of collecting flow-level counters [14], the switch driver can automatically decide how to tune the parameters for these features. If the switch has a trade-off between the counter collection and rule updates, the switch driver can schedule these operations based on the detailed understanding of the trade-off.

Switch drivers send feedback to applications: Sometimes it is impossible for the switch drivers to find a way to meet the application's requirements of the virtual flow tables. For example, when an application requires a VFT with millions of flow-level rules and with the performance requirement of line speed packet processing, the switches can neither install the rules in the hardware table because it is not large enough; nor in the software table because it is not fast enough. When these scenarios happen, the switch driver sends feedback to the application indicating the VFTs that the switch cannot realize. The applications have to modify the VFTs and NOSIX can try to map the new VFTs to the physical tables again. Or applications may rebalance the rules across switches if possible. The feedback is also useful for resource allocation across applications. For example, in order to ensure fairness of resource utilization among VFTs proposed by different applications, NOSIX may send feedback to those applications that require too many resources. We leave the detailed design for supporting multiple applications for future work.

Overhead: The overhead of NOSIX involves (a) keeping the aggregate desired VFT state of the network on the controller, (b) repeatedly computing the mapping between virtual and physical state (c) efficiently transferring the physical state to the switch. As it heavily depends on the exact choice of annotation language, abstraction level and implementation, we leave a quantitative discussion for future work.

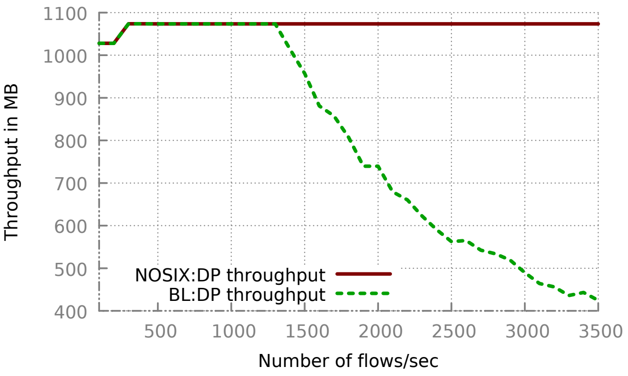

Figure 2: **Throughput with a single flow table**

4. NOSIX CASE STUDY

We provide a preliminary simulation of how NOSIX can enable switch-specific optimization by mapping application expectation and optimization goals with the underlying switch infrastructure. We use two examples to show the benefits for NOSIX to have application knowledge and understand switch design, respectively.

Benefits of having application knowledge: We evaluate our system with an application that creates two VFTs: one with access control rules on elephant flows (e.g., large file downloads) and another with forwarding rules for mice flows (e.g., ARP, DNS queries). Similar to B4 [19], we assume that applications know some properties of their traffic. For instance, in the Google WAN [17] applications are aware of scheduled high-bandwidth traffic when servers are migrated. Even for general Internet traffic, there is correlation between the protocol and TCP port of a flow and its characteristics [21]. The application annotates each VFT with promises on the number of bytes per flow (100 MB for the elephant flow and 10 MB for the mice flow). The application reactively installs a flow entry in the switch for every new incoming flow for both VFTs. We generate a mix of 20% elephant flows and 80% mice flows. The switch has 1 Gbps forwarding rate and the TCAM table has 500 entries. The bandwidth between the controller and the switch is 10 Mbps. We gradually increase the total number of flows until it exceeds the TCAM capacity.

We compare two approaches: (1) Baseline: The switch sends the first packet of each flow to the controller. The application then reactively installs the rule at the switch, so all the following packets can match the rule directly. Once the flow table is full, the switch sends all the new flows to the controller for processing. (2) NOSIX: Since NOSIX knows the elephant and mice flows from VFTs, the switch driver prioritizes flow table entries for elephant flows and redirects all the mice flows to the controller. As shown in Figure 2 and 3, at first, both NOSIX and the baseline install rules reactively based on the incoming flows. Once the flow table is full, the baseline redirects all new flows to the controller irrespective of mice or elephant flows, and thus its throughput is limited by the control channel bandwidth. In contrast, NOSIX removes the rules for mice

[1] In fact, Open vSwitch already added its extensions to the controller-switch interactions in the OpenFlow protocol.

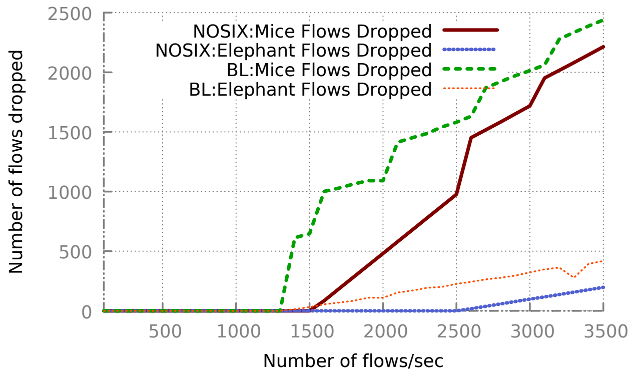

Figure 3: The number of flows dropped with a single flow table

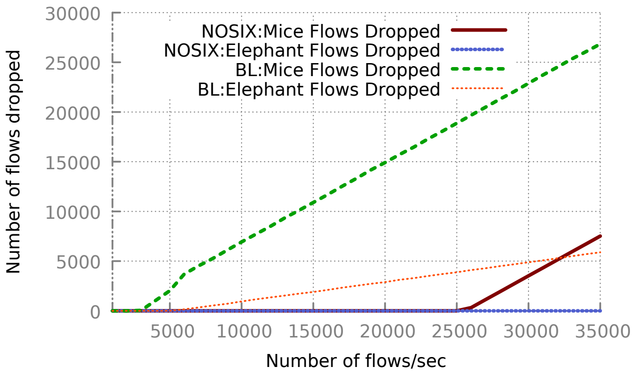

Figure 5: The number of flows dropped with the table pipeline

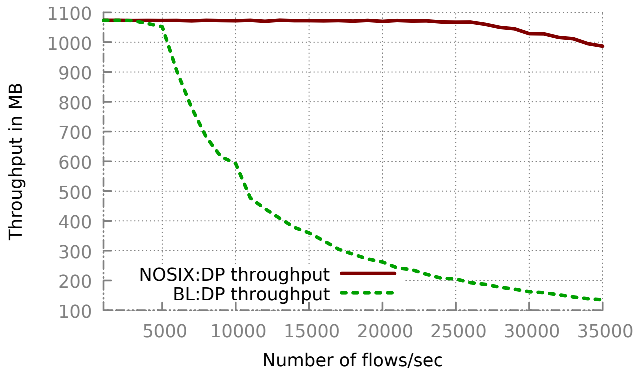

Figure 4: Throughput with the table pipeline

flows and redirects all the mice flows to the controller. As a result, NOSIX achieves higher throughput and drops fewer flows for both mice and elephant flows than the baseline.

Benefits of knowing switch design: We evaluate an application with three VFTs: a reactive L2 access control table, a proactive L2 rewrite table, and a proactive L3 routing table. The switch has an L2 table with 30K entries and a TCAM table with 2K entries.

We compare two approaches: (1) Baseline: The baseline assumes that some switches only support OpenFlow 1.0. In order for the application to be generic across all the switches, the application can only use the single TCAM table. As a result, the switch caches the micro flows in TCAM table and redirects everything to controller. (2) NOSIX: the switch driver leverages the knowledge of switch details and makes use of both tables efficiently. NOSIX merges L2 access control and L2 rewrite rules (VFT1 and VFT2) and map them into L2 hardware table. NOSIX then uses switch TCAM table for the proactive L3 routing. As shown in Figure 4 and 5, the baseline starts dropping flows when total number of incoming flow exceeds the TCAM size of 2k entries. In contrast, NOSIX scales out to 26k new flows. The throughput of NOSIX decreases beyond 26k flows, because the control channel is overloaded and there is no room to send even first packet of incoming flow to the controller for inspection.

5. RELATED WORK

Recent work [16, 29] takes a programming language approach to reducing the complexity of programming SDN. They define declarative or functional languages to enable composition of management modules, reduce the programming complexity, and leverage compilers and runtime systems to optimize the performance. Instead, NOSIX focuses on optimizing the flow-programming pipeline, with the idea that other, higher-level primitives such as Frenetic or the Onix NIB [20] could be built on top of it – in a way an LLVM compiler (Low Level Virtual Machine) of networks. Bosshard [11] proposes a more flexible switch pipeline. NOSIX focuses an approach that works today, where such switches are not yet available. Yanc [23] proposes to leverage operating systems solutions such as file systems to build network controllers. Similar to Yanc, NOSIX also leverages the device driver idea from the operating system. However, NOSIX focuses on supporting portability with diverse switches.

OF-config [9] and the type-length-value (TLVs) in OpenFlow 1.4 allow more syntactic flexibility for SDN messages. NOSIX's switch drivers can be built upon them to support more switch-specific functions and optimization. The Service Abstraction Layer (SAL) in the OpenDaylight project [2] focuses mainly on software engineering aspects of composing software on the controller side, while NOSIX is an abstraction for flow programming in switches. Recent projects that enrich the switch feature set with additional functionality (e.g., OpenFlow Fast-Reroute support, Data-Driven Connectivity [22], Intel Data Plane Development Kit [6]) have highlighted the necessity to continue to innovate on the controller-switch interface. NOSIX can take advantage of those changes in the data plane without exposing them to the application.

Flow Adapter [26] shares the similar idea of mapping a virtual pipeline to the physical pipeline. Flow Adapter takes an algorithmic approach and assumes the switch consists of a series of TCAM-based tables. In reality, hardware switch pipelines consist of many specialized-purpose tables. Therefore, NOSIX leverages the annotated virtual flow tables to figure out how to make use of those special-purpose tables in different switches to improve the resource utilization.

6. SUMMARY

Programming software-defined networks is challenging in the presence of a heterogeneous switch landscape, and applications have to choose between portability and efficiency. This is not a short-term phenomenon, as ASIC Design involves complex cost trade-offs and incurs long product cycles. As such, there is a natural tension between the dynamics of what the software side expects and the hardware can provide.

We propose NOSIX, a lightweight portability layer, which allows application to express their expectations on virtual flow tables, while relying on the vendors to build switch drivers to optimize for specific switch implementations. The approach is indented to be practical even in today's highly diverse and limited ASIC landscape. As such, it does not attempt to isolate the application entirely from the switch. Instead, it provides a lever that enables applications to specify details that they care about and omit ones that are not important. We expect the annotation language to grow in expressiveness and abstraction level over time. As part of our ongoing work, core aspects of NOSIX are currently being integrated into a production enterprise-grade controller framework.

Acknowledgements

We thank our shepherd Marco Canini and the anonymous reviewers for their detailed comments. We also thank Jennifer Rexford and Kyriakos Zarifis for discussions about ideas in this paper. Andreas' work was partially funded through a DAAD FIT research scholarship and by Big Switch Networks.

7. REFERENCES

[1] `http://noviflow.com/products/noviswitch/`.

[2] `www.opendaylight.org`.

[3] Cisco Nexus 1000V. `http://www.cisco.com/en/US/products/ps9902/index.html`.

[4] The complexity of hardware openflow switches. `http://www.youtube.com/watch?v=RRiOcjAvIsg`.

[5] IBM System Networking RackSwitch G8264. `http://www-03.ibm.com/systems/networking/switches/rack/g8264/`.

[6] Intel SDN switch specification. `http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ethernet-switch-fm6000-sdn-paper.pdf`.

[7] NEC ProgrammableFlow PF5240 Switch. `http://www.necam.com/SDN/doc.cfm?t=PFlowPF5240Switch`.

[8] Open networking foundation product directory. `http://sdndirectory.opennetworking.org/products`.

[9] OpenFlow management and configuration protocol. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.1.pdf`.

[10] OpenVSwitch. `http://openvswitch.org/`.

[11] P. Bosshart, D. Daly, M. Izzard, N. McKeown, J. Rexford, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. Programming protocol-independent packet processors. In *arkiv*, 2013.

[12] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *SIGCOMM*, 2013.

[13] M. Canini, P. Kuznetsov, D. Levin, and S. Schmid. Software transactional networking: Concurrent and consistent policy composition. In *ACM HotSDN*, 2013.

[14] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. In *SIGCOMM*, 2011.

[15] N. Egi, A. Greenhalgh, M. Handley, M. Hoerdt, F. Huici, and L. Mathy. Towards high performance virtual routers on commodity hardware. In *Proc. ACM CONEXT*, 2008.

[16] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: a network programming language. In *Proc. ACM SIGPLAN ICFP*, 2011.

[17] U. Hölzle. OpenFlow at Google. Open Network Summit, Keynote. `http://www.youtube.com/watch?v=VLHJUfgxEO4`, 2012.

[18] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. *ACM HotSDN*, 2013.

[19] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. In *SIGCOMM*, 2013.

[20] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: a distributed control platform for large-scale production networks. In *USENIX OSDI*, 2010.

[21] Y.-S. Lim, H.-C. Kim, J. Jeong, C.-K. Kim, T. T. Kwon, and Y. Choi. Internet traffic classification demystified: On the sources of the discriminative power. In *CoNEXT*, 2010.

[22] J. Liu, A. Panda, A. Singla, B. Godfrey, M. Schapira, and S. Shenker. Ensuring connectivity via data plane mechanisms. In *NSDI*, 2013.

[23] M. Monaco, O. Michel, and E. Keller. Applying operating system principles to SDN controller design. In *HotNets*, 2013.

[24] M. Moshref, M. Yu, A. Sharma, and R. Govindan. Scalable rule management for data centers. In *NSDI*, 2013.

[25] OpenFlow Wiki: NEC Univerge PF5820. `http://www.openflow.org/wp/switch-nec/`.

[26] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie. The FlowAdapter: Enable flexible multi-table processing on legacy hardware. In *ACM HotSDN*, 2013.

[27] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *HotNets*, 2011.

[28] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore. OFLOPS: An Open Framework for OpenFlow Switch Evaluation. In *Proc. PAM*, 2012.

[29] A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *PADL*, 2011.