NOSIX: A Portable Switch Interface for the Network Operating System

Andreas Wundsam
ICSI

Minlan Yu
USC

ABSTRACT

Even with the success of software-defined networks (SDN), it remains challenging to write a *portable*, *correct*, and *efficient* controller application. This is due to the mismatch between the expectations of controller applications and (i) the heterogeneity of switches and (ii) the primitives offered by OpenFlow, the prevalent SDN protocol. We introduce NOSIX, a portable low-level switch programming API. Like its OS name-cousin, it facilitates the development of portable low-level applications (i.e., applications that directly built on the flow model) and at the same time serves as a building block for higher abstraction frameworks [12, 10]. Designed to be a minimalistic model, NOSIX purposefully only targets one switch as a time, leaving network wide consistency, topology and distribution to higher layers. It provides applications with a view of *virtual flow tables* without resource or feature constraints, while supporting consistent and efficient rule updates and counter collection. Underneath NOSIX, vendor defined *switch drivers* synchronize the virtual flow-tables with the physical tables on the switch, and thus hide heterogeneity and enable switch-specific optimization.

1. INTRODUCTION

The software-defined network (SDN) paradigm, currently successful in both academic [2, 11] and commercial environments [5, 3], gives operators the opportunity to program the control plane of their networks, with the ubiquitous OpenFlow network API [13] serving as the linking element of the ecosystem.

While OpenFlow enables control plane programmability, it is widely acknowledged that it is challenging to write *portable*, *correct*, and *efficient* applications[1] on top of OpenFlow [19, 6, 16]. This is due to the mismatch between application expectations and (i) switch heterogeneity (ii) the primitives offered by OpenFlow. Ideally, an SDN application programmer *expects* to generate rules with unbounded flow tables, a full OpenFlow feature set, and high bandwidth in the forwarding plane; the programmer also wants to correctly and efficiently update rules and collect counters based on certain data plane or control plane events. However, in reality, she may have to deal with non-idempotent primitives like `FLOW_MOD`, `FLOW_EXPIRE`, handle spurious `PACKET_IN` messages, and emulate consistency through under-specified synchronization primitives like `BARRIER`. At the same time, OpenFlow switches differ significantly in architecture (e.g., hardware vs. software), feature set (e.g., flow table sizes, supported rewrites), and performance (e.g., fast vs. slow path) and their interpretation of the specification (e.g., for the `BARRIER` command) [17]. While many high level controller frameworks aim to solve the diversity problem, they restrict the programming model, and don't lend themselves to building low-level applications "close to the metal". Consequently, each controller framework or application has to be adopted to specific switch design to work correctly and perform optimally, resulting in duplication of effort and a combinatorial explosion of customization. Not every developer will have the resources or expertise to adapt their system to every switch, resulting in sub-optimal performance and functionality.

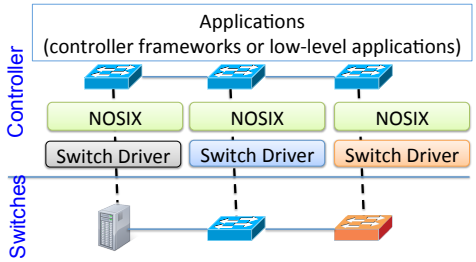

Figure 1: NOSIX

We propose to focus on operations on the *switch*, separate the application expectations from the switch implementation specifics, and create an interface that enables *portability* while preserving *efficiency* and *switch-level expressiveness*.

In contrast to previous work on powerful, abstracted programming models for the entire network [12, 9, 10], we introduce a new low-level API called NOSIX[2], as

[1] In this paper, we use applications to denote both controller frameworks and the low-level applications built directly on OpenFlow-like flow model.

[2] In allusion to the well known POSIX standard that defines

a *minimalistic* API optimized for low-level applications "close to the metal" and for building other controller frameworks (Figure 1). Thus, we focus on primitives that are (i) *local*, i.e., that can be implemented on a single switch, (ii) *ubiquitous*, i.e., that are required by most applications, and (iii) *expressive*, i.e., that don't restrict the expressiveness of OpenFlow.

At the core of our API, we define the *virtual flow table*, which is an idealized flow table that has no resource constraints and full feature-set. It represents the application's view of the currently active flow-level rules. NOSIX supports *consistent updates* of the virtual flow table (i.e., no invalid intermediate states are exposed) and collects *counters* efficiently from the switch. Underneath NOSIX, vendor-defined *switch drivers* in the controller map the virtual flow table to the physical flow tables of the switches, and thus hide the switch diversity and perform switch-specific optimization.

NOSIX facilitates writing portable SDN applications through its simple model of an *idealized* OpenFlow switch, which is expressive enough to be amenable to low-level tasks. NOSIX is useful as a building block for other high-level controller frameworks, reducing duplication of efforts in programming diverse switches. Also, moving the common switch API from the switches up to the controller opens up space for optimization below, as vendors can utilize the specific strengths of their forwarding hardware and optimize the controller-switch communication protocol.

In the remainder of this paper, we first investigate the switch heterogeneity and the resulting challenges of programming a single switch (§2). We then present our suggestion for a low-level API for switch programming NOSIX in §3. Then, we discuss related work in §4 and conclude in §5.

2. CHALLENGES OF PROGRAMMING A SINGLE SWITCH

One key building block for programming SDN is to program a single switch. Ideally, this should be a simple task enabled by OpenFlow. However, in practice, switch implementations differ substantially in both the forwarding plane and the control plane. There are hardware switch implementations based on different platforms (e.g., from HP, NEC, Quantas), which differ in their features and performance [17]. In contrast, there are software switches [1] that have fundamentally different trade-offs. This diversity is insufficiently hidden and abstracted by the primitives offered by OpenFlow. We now discuss some of the differences in the forwarding and control plane of the switches and the challenges they entail.

a low-level OS API that enables portability for OS programs, and serves as a building block for high level platforms.

2.1 Forwarding plane: resources, features and performance

Heterogeneity: The forwarding plane forwards packets according to the rules in the flow table. An ideal forwarding plane would provide an unlimited number of flow tables of unbounded size, support the full OpenFlow feature-set, and provide full bandwidth for all forwarding. However, in practice, the forwarding planes differ significantly depending on their internal forwarding architecture. Hardware forwarding planes are implemented in discrete logic, using TCAM and hardware hash tables, and their features depend on the switching fabric that is used. Software forwarding planes have fewer fixed resource constraints and more complete feature sets, but their performance is more unreliable.

As a result, forwarding planes are highly heterogeneous in three aspects: *(1) Resource constraints:* Hardware switches differ in the size and number of exposed forwarding tables. E.g., the HP ProCurve 5406zl supports 1,500 flows [7], while the NEC PF5820 supports 750 [14]. The software forwarding plane in Open vSwitch supports a conceptually unbounded number of rules. *(2) Feature set:* There are also differences in the flow fields and the number and type of the actions that can be supported. For instance, one of three hardware switch evaluated by Rotsos [17] did not support IP address rewriting. Open vSwitch supports the full set of matching and actions, and the number of actions is only limited by OpenFlow's 64 KByte limit [4]. *(3) Performance:* Hardware switches provide data-plane forwarding at full line-rate, independent from the switch CPU and host bus. In contrast, the data-plane forwarding speed of a software switch depends on the system speed, software optimization, and may deteriorate with growing flow table size [17, 8]. In hardware switches, features not directly supported by the hardware may be implemented in the firmware on the *slow path*, several orders of magnitude slower. Software switches typically don't have the concept of fast and slow paths.

Programming challenges: These limits and differences in the forwarding plane presents two key challenges in programming:

The number of rules applications generate vs. flow table size: An access control application may require a large number of fine-grained rules, more than the flow table size at the switch. The application has to keep the rules local at the controller and only installs the most active rules in the switch, while emulating the other rules through `PACKET_IN` events in the controller.

Features vs. performance: Applications may desire full features *and* optimal performance, however, in many cases there is a trade-off between the two, as features not directly implementable in hardware can be emulated in firmware. The application has to understand the performance trade-off of using each feature and decide which

one to use. For example, if the switch has TCAMs, the application can proactively install a wildcard rule; otherwise, the application may choose to reactively cache microflow rules.

2.2 Control plane: updates and counters

The switch control plane processes *rule updates* from the controller to the forwarding plane, and delivers flow-level *counters* to the controller. An ideal switch control plane would be able to handle an unbounded number of flow updates, taking effect instantaneously. The rules in the switch flow table would be predictable by the controller. However, in practice, there are limits imposed by switch diversity, the semantics of OpenFlow primitives, and the control channel, making it hard to program applications correctly and efficiently.

Heterogeneity in rule updates: Rule updates differ at switches in two aspects: *(1) Performance:* In hardware switches, modification and state queries of the forwarding plane are expensive and the modification rate is limited [17]. Thus, updates are scheduled, this causes delays in flow installation and may lead to inconsistencies. In software switches, changes are quicker, but may still be delayed through the OF control channel. *(2) Consistency primitives:* The `FLOW_MOD` commands may be delayed or reordered. The `BARRIER_REPLY` message has different semantics in different switches: the software switch sends the message after the rules are updated at the switch, while hardware switch with TCAMs send the message after the rule updates are scheduled but not necessarily effective in TCAMs.

Programming challenges in rule updates:

Flow-table synchronization: To effectively compute forwarding updates, and enable fail-over between switches, applications have to know the existing rules at the flow tables. Therefore, applications often keep a local copy of the flow table and *synchronize* it to the switch. However, this is non-trivial because switches may automatically expire some rules and may delay `FLOW_MOD` commands. When control channel is temporarily interrupted and recovered, the application no longer knows the state of the switch flow table. The most common practice is to clear the switch flow table in this case, and repopulate it from scratch. While this 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 new rules. 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.

Consistent updates: Naively updating a group of rules in a flow table can lead to incorrect transient states. Imagine a firewall middlebox configured to intercept traffic passing through an ingress switch. At some point, the operator decides to migrate the firewall to a more powerful system on a different port due to increased traffic. The configuration change thus consists of two updates *delete(ingress→ p_1)* and *install(ingress → p_2)*. If these are executed in the wrong order and without *per-packet consistency* [16] guarantees, traffic may pass into the network unfiltered during the update. It is also non-trivial to build network-wide consistency guarantees on top of `BARRIER` [16], especially when the `BARRIER_REPLY` is implemented differently.

Heterogeneity in counter collection: The controller can use the `FLOW_STATS` command to query the current flow-level counters at the switch. In TCAM based switches, querying counters for the entire flow table is an expensive operation. Thus, switches have different design decisions on how often these counters can be gathered without interfering with other control plane tasks. Some switches have inadequate scheduling of `FLOW_STATS` queries, leading to performance bugs that are hard to identify [19]. Others provide custom optimizations (e.g., sampling, approximate counters as proposed in DevoFlow [7]).

Programming challenges in counter collection: Depending on the performance of querying individual rules and the entire table, the application has to optimize which counters to query and how often to query them. For switches with more efficient collecting features [7], the developers have to understand and design their programs based on these features.

2.3 Switch heterogeneity is inevitable

We believe switch heterogeneity is *not* a short-term phenomenon, because the software side of SDN evolves faster than the switch platforms, and there are different target markets and architecture trade-off.

Some feature and performance differences among switches stem from the fact that OpenFlow is not yet mature. The first generation SDN prototype switches of most switch vendors have been built on top of their existing switch fabrics, limiting the achievable feature set. Currently, the SDN software and OpenFlow protocol keep evolving rapidly, making it harder for the hardware to keep up. Thus new features (e.g., multiple tables, group tables in OpenFlow 1.1) are as yet rarely supported by the switch implementations. We expect this situation to change only gradually, and that this evolution-based diversity will persist for several years until the SDN protocols and implementations become mature.

In the long term, switch heterogeneity will still exist because of two reasons: (1) There will be switch implementations with different trade-offs of cost and scale. For example, software switches often have richer features than hardware switches but with worse performance. (2) These switches will be customized for different applications and markets. and routers, there will still be many different switch implementations in the future.

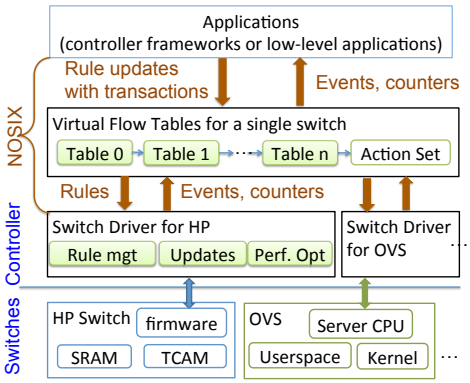

Figure 2: NOSIX architecture

3. NOSIX: A PORTABLE SWITCH API

We propose NOSIX, a portable API to program a single Switch by separating application expectations from switch heterogeneity. In this section, we first discuss our high-level design decisions in achieving the right trade-off between portability and efficiency. Next, we describe how to separate application expectations from switch-specific designs. As show in Figure 2, we have two key designs to enable NOSIX: the *virtual flow tables* that represent the application's expectation of the rules in an ideal switch; and the *switch drivers* written by vendors in the controller that enable switch-specific customization and optimizations.

3.1 Trade-off of portability and efficiency

The key challenge of programming SDN applications on heterogeneous switches is to strike a right balance between the portability and efficiency. If the application does not consider specific switch implementations, it is portable but may be inefficient. However, to optimize the performance of an application, the programmer has to write different code customized for each kind of switches. To achieve the right trade-off between the portability and efficiency, we make two design decisions:

Applications expose the expectations to the switch: One way to handle switch heterogeneity is to expose switch features to a runtime system (e.g., Frenetic [10]), which automatically transforms the rules the application generated to the new rules that match best to the switch implementation. For example, if the switch supports wildcard rules, the application may proactively generate the wildcard rules; if the switch does not support wildcard rules, the application may reactively cache micro flow rules at the switch. While the runtime system reduces the complexity of individual applications, it is still challenging for the switches to expose detailed feature set (e.g., how `BARRIER` is implemented) and performance trade-offs (e.g., how the switch schedule updates and counter queries) to the runtime system. It is also challenging for the runtime system to handle the different switch dynamics in updating rules, generating notifications, reporting flow-level counters without detailed understanding of the switch design.

Instead of exposing the switch heterogeneity to the upper layers (applications or the runtime system), we argue that the applications should specify their expectations of the rules, updates, and notifications to the switch as if it has an ideal switch. It is then switch vendors' job to choose the best way to implement these expectations, because vendors are in the best position to optimize for their own switches. Therefore, we introduce a simple abstraction *virtual flow tables*, where applications can freely define the rules without worrying about either the resource constraints and feature set in the forwarding plane, or the delay and throughput of updates and notifications.

Vendors program switch drivers in the controller: When the applications expose their expectations to the switch, now the complexity moves to the switches. Unfortunately, switches are notoriously hard to program and have poor CPUs. Therefore, we argue that vendors should implement *switch drivers* in the controller to hides switch heterogeneity from applications and enable diversity and innovations of both the switch forwarding plane and control plane. In the forwarding plane, instead of following the same specification, vendors now have the flexibility to design their own features and resources (e.g., the types and number of flow tables), as long as they can use the driver to manage the rules based on applications' expectations.[3] In the control plane, vendors can design different mechanisms to process updates and send notifications to the applications, as long as they follow the same minimalist abstraction of application's expectations.

By implementing switch drivers at the controller, we actually move part of the *switch control plane* to the controller (i.e., closer to the applications). This unlocks a large potential for optimization: (1) The control channel between the controller and the switch may have delay churns and failures, leading 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[4]. (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 forwarding plane, the firmware, or the switch driver. (3) A single driver in the controller can manage

[3]Note that today some vendors already use firmware to mimic OpenFlow features they cannot support in their hardware (e.g., packet encapsulation). With switch drivers in the controller, vendors have more flexibility to decide whether to support a function in hardware or not.

[4]In fact, Open vSwitch already added its extensions to the OpenFlow protocol

all the switches belonging to the same vendor, enabling more switch-specific optimizations.

3.2 Separate application view from switches

NOSIX separates the application's expectations from the switch-specific implementations. We design and build two key components as shown in Figure 2: (1) A pipeline of multiple 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. These flow tables are then connected into the pipeline.[5] (2) The switch drivers that provide switch-specific optimizations based on the applications' expectations. Each switch driver is responsible to transform the rules in virtual flow tables into the actual switch flow tables in the forwarding plane. The driver also updates the switch flow table according to the changes of virtual flow tables, and propagates switch events to the applications. We now discuss three design aspects of separating application expectation from switch implementations:

Flexible mappings between virtual flow tables and physical switch tables. These virtual flow tables do not need to have a one-on-one mapping with physical flow tables. For example, the application may define one table for measurement rules and another for forwarding rules, and the switch driver may merge and map them to the single TCAM table in a hardware switch.

The rules in the virtual flow tables do not need to be always in the physical flow tables. For instance, the application can define a single table of 50K access control rules, but the TCAM table in the switch may only support 10K rules. The device driver caches 10K rules in the TCAM and dynamically update the cached rules based on the `packet_in` event. Since these packets experience cache miss at the switch but matches rules in the virtual flow table, they are handled by the device driver and not propagated to the applications.[6]

The rules in the virtual flow tables do not need to be of the same format with those in the physical flow table. For example, suppose an application *proactively* installs a *wildcard* rule. If the switch does not have TCAM to support wildcard rules, the switch driver can choose to *reactively* cache the corresponding *microflow rules* in the switch based on the wildcard rule. If the switch has TCAM but supports rule cloning (as proposed in DevoFlow [7]), the switch driver may also leverage the feature and proactively install the rules.

[5]The pipeline is similar to the pipeline of flow tables defined in the OpenFlow 1.1 specification. The key difference is that these virtual flow tables are located at the controller instead of the switches and do not have any resource constraints.

[6]In contrast, the switch driver simply passes the `packet_in` event to the application if there is no rule to match the packets in the virtual flow tables either.

Since there are not any constraints on the mappings between the virtual flow tables and the physical ones, applications can freely generate the rules based on their design.

Separation between rule update semantics and the update mechanisms at switches: When an application modifies the rules in the virtual flow table, the device drivers should update the rules in the switch accordingly. Applications requirements may dictate that such changes are applied with selectable consistency, i.e., with no intermediate states exposed. Switches differ in their support and primitives offered for updates and consistency. We propose to separate the update semantics of applications from update mechanisms at switches.

To represent the application's expectation on rule updates, we propose a simple transactional semantics with two primitives: *start_transaction(consistency_level)* and *commit(wait)*. All the operations the applications specify on the virtual flow tables after *start_transaction* should remain invisible to the underlying switches until the *commit* command. Then, these operations are executed based on the selected *consistency_level* (e.g., no consistency, per-packet consistency, and per-flow consistency [16]). The *commit* command returns based on the *wait* parameter (e.g., return immediately, return until the change is effective, and return when the switch ensures the following updates are scheduled after the current one).

The switches may have different ways of achieving the consistency. Consider the firewall rule example in Section 2.2. The application removes the rule (ingress$\rightarrow$ p_1) and installs a new rule (ingress$\rightarrow$ p_2), requiring per-packet consistency. The switch drivers collect the modifications of the virtual flow tables (Δ) and provide different ways of enforcing the Δ at switches. If a switch natively supports packet-consistent updates, its driver would simply pass through the commands. To work with a standard OpenFlow switch, its driver could determine that a packet_consistent order of these operations: installing the new rule before deleting the old one.

The switch driver have to implement *wait* settings in a custom way. Suppose an application expects the commit to return after the rules are effective at the switch. Note that there is no generic OpenFlow way to implement this wait_state *effective*: The driver *must* use its specific knowledge of the switch to ensure the configuration is active when it returns. For instance, if the switch is known to the driver as a soft switch with immediate flow scheduling, the `BARRIER_REPLY` will indeed signify the rules are active. For a hardware switch, `BARRIER_REPLY`s do not offer that guarantee, the driver may use a vendor extensions or fall back on knowledge of TCAM scheduling (e.g., flows get active in the TCAM at most 2 seconds after the `BARRIER_REPLY` is issued) to wait for an upper bound of the scheduling delay before continuing.

Although our transaction semantics and customized

switch drivers focus on a single switch, they are useful as building blocks that simplify implementing the network-wide consistency properties defined in [16].

Decoupling of application-specific and switch-specific performance optimization of counter collections: The performance of collecting counters from the switch to the virtual flow table depends on both the application design and the switch implementation. To simplify the interactions between virtual flow tables and switch drivers, we propose to separate the application-specific performance optimizations from the switch-specific ones.

The applications can optimize the performance based on their design logic and performance requirements without generic switch assumptions (e.g., the fewer queries, the better.). For example, the runtime system in the controller can merge the flow-level measurement rules from different applications into a single one in the virtual flow table [10]. The application can also choose to only collect counters from a limited number of important flows if they find it expensive to collect many counters at a time.

Next, the switch driver can optimize the performance based on the switch feature set and the trade-off of feature and performance. For example, if the switch supports more advanced ways of collecting flow-level counters [7], 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 (see Section 2.2), the switch driver can schedule these operations based on the detailed understanding of the trade-off.

Although we may not achieve the optimal performance with the separated optimizations, we can still highly improve the performance because vendors have better knowledge of switch implementations. We will compare the separated optimizations to the joint optimization with both application and switch knowledge in future work.

4. RELATED WORK

There have been many works on improving the programming of software-defined networks. Onix, POX, and Floodlight [12, 9, 15] provide a layered architecture, by assuming applications to be logically centralized and work on an abstracted graph of the network. Frenetic and Nettle [10, 18] take a programming language approach to reducing the complexity of programming SDN. They define a declarative or functional language to enable composability and reduce the complexity, and leverage a runtime system to optimize the performance. In the OS analogy, these controllers are like high-level frameworks like JVM and PostgreSQL. They require complex runtimes that handle the idiosyncrasies of the OpenFlow programming model and the diversity in switch designs. Complementary to these works, NOSIX focuses on programming a single switch and hiding switch heterogeneity. NOSIX provides a simple abstraction (like POSIX) to separate the applications requirements from switch heterogeneity and introduces vendor-defined switch drivers for switch-specific customization and optimization.

5. SUMMARY

Programming a single switch is a key component to program the network. Yet, this is surprisingly difficult when the switches are heterogeneous, and applications have to choose between portability and efficiency. Instead of exposing full switch capabilities to the applications, NOSIX allows application to express their expectations on virtual flow tables, while relying on the vendors to build switch drivers at the controller to optimize for specific switch implementations. NOSIX opens many switch-specific optimizations while keeping the applications simple to program.

6. REFERENCES

[1] http://openvswitch.org/.

[2] EU Project Ofelia. http://http://www.fp7-ofelia.eu/.

[3] Nicira Networks Inc. http://www.nicira.com/.

[4] Ben Pfaff. Email on the ovs-discuss mailing list, 23 Mar 2012. http://openvswitch.org/pipermail/discuss/2012-March/006759.html.

[5] BigSwitch Networks. http://http://www.bigswitch.com.

[6] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE Way to Test OpenFlow Applications. In *USENIX NSDI*, 2012.

[7] 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.

[8] 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.

[9] Floodlight Controller. http://floodlight.openflowhub.org/.

[10] 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.

[11] GENI: Global Environment for Network Innovations. http://www.geni.net.

[12] 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.

[13] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *CCR*, 2008.

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

[15] POX - An OpenFlow Controller. www.poxrepo.org.

[16] M. Reitblatt, N. Foster, J. Rexford, and D. Walker. Consistent updates for software-defined networks: Change you can believe in! In *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ACM HotNets Workshop, 2011.

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

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

[19] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann. OFRewind: Enabling Record and Replay Troubleshooting for Networks. In *Proc. USENIX ATC*, June 2011.