FOCUS: Function Offloading from a Controller to Utilize Switch Power

Technical Report CS-TR-2016.001
Department of Computer Science, Duke University
February 2016

Ji Yang *
Xi'an Jiaotong University and Duke University

Zhenyu Zhou *
Duke University

Theophilus Benson
Duke University

Xiaowei Yang
Duke University

Xin Wu
Big Switch Networks

Chengchen Hu
Xi'an Jiaotong University

Abstract

Software Defined Networking (SDN) uses a logically centralized controller to replace the distributed control plane in a traditional network. One of the central challenges faced by the SDN paradigm is the scalability of the logical controller. As a network grows in size, the computational and communication demand faced by a controller may soon exceed the capabilities of a commodity server. In this work, we revisit the task division of labour between the controller and switches, and propose FOCUS, an architecture that offloads a specific subset of control functions, i.e., *stable local functions*, to the switches' software stack. We implemented a prototype of FOCUS and analyzed the benefits of converting several SDN applications. Due to space restrictions, we only present results for ARP, LLDP and elephant flow detection. Our initial results are promising and they show that FOCUS can reduce a controller's communication overhead by 50% to nearly 100%, and the computational overhead from 80% to 98%. Furthermore, we observe that FOCUS offloading to the switches saves switch CPU because FOCUS reduces the overheads for communication with the controller.

1 Introduction

Software Defined Networking (SDN) is a new paradigm that introduces a logically centralized controller to replace the distributed control plane in a traditional network. The benefits of employing SDN over traditional networking include: ease of network management, ease of development, and ease of adoption of new protocols. Modern SDN controllers are used to run a variety of SDN applications ranging from topology management and host management protocols to traffic engineering, security, and cloud virtualization applications. Unfortunately, the SDN controller introduces several critical challenges including scalability: SDN controllers are unable to scalably process events from a network with hundreds or thousands of devices (switches and end-hosts) [26].

Existing proposals to address this challenge fall broadly into two categories. The first category of solutions delegate certain functions from a controller to the switches to reduce a controller's work load. Examples include introducing switch primitives such as rule cloning, local actions [11], or stateful transitions [20,34], in-switch packet generation [10], and turning on legacy protocols on a switch [21]. The other category of solutions [8,13,16,28] uses a distributed controller architecture that distributes a controller's functionality and load among multiple physical machines. This work falls into the first category of solutions. We aim to reduce a controller's work load by delegating some of a controller's functions to switches. It complements the distributed controller architecture as distributed controllers can also delegate functions to switches to reduce the load on each controller.

A key design decision to make in developing a delegation architecture is how to tradeoff the complexity and cost of delegating a function with the benefit of the delegation.

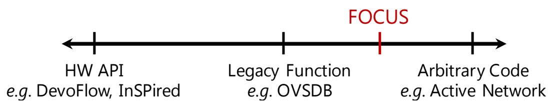


Figure 1: **Design space for delegating control to switches.**

At one end of the spectrum (Figure 1), one can design a programming environment that allows a controller to offload a piece of code to switches. This design offers the flexibility of controlled delegation and enables new control functions, but it requires a common code execution environment on all switches and may bring other unexpected performance and security issues [27]. At the other end, one can add a small set of flexible hardware

*These authors contributed equally to this work primitives to the switches. This design offers limited flexibility but provides the best performance. Unfortunately, it requires modifications to hardware switches – a feat that is not easily achieved. Thus today, most datacenters explore a middle ground by turning on the legacy protocols on switches. A controller, instead of running a control application, uses the OVSDB [21] protocol to obtain control information from switches. This design has the advantages of requiring no additional development cost on switches, but is only limited to legacy protocols that already exist in switches – new protocols require the cumbersome development effort to implement them. It also gives a controller little control over what functions to delegate, as it is an all-or-nothing solution: a controller either delegates all functions provided by a legacy protocol to switches or nothing at all.

In this work, we aim to strike a balance between the flexibility and effectiveness of delegation and the cost of delegation. We chose to delegate the control functions that have the following two characteristics. First, the output of those functions only depends on input local to a switch. Second, the output remains relatively stable as long as the network configuration does not change. We observed roughly two classes of applications that fit this bill. The first, a set of applications that force the controller to periodically poll switches for information. For example, traffic-engineering [5, 7, 12, 25], load-balancers [33], and security applications [6, 24, 25, 32]. The second, a set of applications that force the switches to periodically query the controller for information, e.g., ARP and ICMP.

We propose the FOCUS (Function Offloading from a Controller to Utilize Switch power) architecture, where a controller delegates these stable local functions to switches. We are attracted to delegating stable local functions because we can offload a variety of those functions to switches with a simple set of APIs (less than nine APIs calls) without introducing switch-to-switch communications and without introducing a code execution environment at switches. The examples we explore in this work include a number of packet and traffic report generation functions such as ARP replies, ICMP responses, LLDP's link discovery function, DHCP relays, and elephant flow detection. Due to space limit, we only focus on three such applications: LLDP, ARP, elephant flow detection. From our empirical knowledge of SDN networks, we believe these functions constitute as some of the most commonly used protocols in practice.

Unlike existing techniques [11, 20, 31, 34], FOCUS only delegates control plane functions to a switch's software stack, and does not require changes to a switch's hardware. We made this design decision for simplicity of design and ease of deployment. Different from [10], which only delegates in-switch packet generation functions ARP and ICMP, FOCUS has the flexibility of delegating more functions such as periodic traffic report to switches.

As a proof of concept, we implemented and evaluated the controller's communication and computation workload before and after function offloading for three functions: ARP replies for the default gateway, LLDP's link discovery function, and elephant flow detection using the FOCUS architecture. Our experiments show that function offloading can reduce a controller's CPU utilization by 98.5%, 97.7%, over 80%, for ARP, LLDP, and elephant flow detection respectively, and reduce the number of protocol messages between the controller and switches by almost 100% for ARP and 50.0% for LLDP. In addition to the three functions we implemented, we examined and analyzed several other commonly used SDN applications and showed tremendous savings with FOCUS (we omitted these application due to space constraints).

Further, while our original design goal was to reduce a controller's utilization, our experiments revealed that delegating stable local functions can also reduce a switch's utilization. This is because offloading reduces the number of control messages a switch sends to and receives from a controller. When the computational savings in I/O processing offset the amount of computation a switch performs to carry out the offloaded functions, the overall workload on a switch also reduces.

In summary, the main contributions of this paper are as follows:

- The design of the FOCUS architecture, a function delegation architecture which significantly reduces SDN controller's overhead and makes it more scalable.

- A set of simple APIs that can enable a variety of control function delegation. A switch implements the APIs, and a controller uses this API set to delegate control functions.

- The implementation and evaluation of three commonly used control functions including ARP replies, LLDP link discovery, and elephant flow detection. The experimental results show FOCUS can significantly reduce both a controller's and switches' workload.

2 Background and Motivation

In this section, we present a brief overview of switching design, discuss commonly used applications, and design requirements for a successful new framework.

2.1 OpenFlow Switch Design

Packet processing on OpenFlow switches occurs in roughly two components, the ASICs, and the CPU.

The ASICs embodies the hardware, TCAM, and SRAM, which enables line-rate forwarding and matching (often considered the fast path). Traditional flow table entries and rules are stored in TCAM or SRAM depending on the switch configuration.

The CPU runs the Switch Operating system consisting of drivers for controlling the ASICs and an agent for communicating with the controller. The OpenFlow agent maintains a TCP connection to the controller, exchanges OpenFlow messages with the controller, and translates controller messages into hardware instructions for the driver (often considered the slow path). The CPU processes packets at about two to three orders of magnitude less than the ASIC. This difference in processing speeds makes the CPU inadequate for line rate processing but sufficient for occasional packet processing.

2.2 Motivation

In this section, we discuss three classes of application run in modern SDN networks and use these classes of applications to motivate a novel set of switch abstractions. The first class of applications, Host Discovery, are used by hosts to discover services and debug network problems. Examples of these includes ARP and ICMP. The Second, Topology Maintenance, are used by the controller to discover and maintain accurate topology information. The last class, Control Applications, are control plane applications that aim to improve network performance, security, availability and other features by using traffic statistics to inform routing decision.

ARP (Default Gateway): In a network, whenever a host tries to communicate with another host outside of its Local Area Network, it needs to send the packets to its default gateway. To do this, the host must first send an ARP request for the gateway's MAC address.

In an SDN network (Figure 2 (a)), these ARP requests are received by the host's edge switch and forwarded to the controller. The controller, in turn, looks up the MAC address and returns it to the edge switch, which returns it to the end host.

Observation: The IP and MAC of the default gateway are predefined: it is constant and rarely changes. Further, all hosts sharing the same edge often share the same gateway. Thus, the response for all ARP requests for these end hosts will be identical. Today all ARP requests have to be sent to the controller. From this, we observe that there is an opportunity to delegate the process of generating ARP replies. Moreover, offloading does not violate the visibility of the controller because we are essentially caching the controller's responses. Further, many applications, e.g. ICMP-Echo, ICMP-Timeout, IGMP, and DHCP Relay, run within modern SDN networks have a similar pattern.

Local Link Discovery Protocol (LLDP): Networks use LLDP to discover the physical links. In an SDN network (Figure 2 (c)), the controller generates a unique LLDP packet for each port and instructs the switch to send the LLDP packet through the port to the device at the other end of the link. The receiving device, in turn, sends the LLDP packet back to the controller. Upon recipient of this LLDP packet, the controller learns of the existence of the physical link connecting the two devices. To maintain an accurate and up-to-date topology, the controller must periodically send these LLDP packets.

Observation: These periodic packets incur a tremendous amount of overhead and are used by other topology maintenance protocols including LACP and BDDP. Interestingly, the only time these periodic packets include any new information is when they are used to detect link failures. From this, we observe another opportunity to delegate functionality to the switches. Namely, the process of periodically generating packets. Similarly, we note that this does not violate the visibility of the controller provided that the switch generates a report to the controller when: (a) the port is down, and the switch can not generate and send the periodic packets and (b) when the switch does not receive the periodic packets from the other end of the link.

Elephant Flow Detection: Many SDN applications [5–7, 12, 24, 25, 32] employ packet counters maintained by a switch to measure, monitor, secure and improve the performance of the network. From this, most of these applications periodically poll the network switches for this information. For example, Hedera [5] (Figure 2 (e)), a traffic engineering application for data centers, aims to improve performance by detecting elephant flows, large flows, and placing elephant flows on distinct links. To detect elephant flows, the SDN controller has to periodically query every switch for packet counters.

Observation: Similarly to the previous example, the elephant flow pattern incurs a tremendous amount of overheads. Interestingly, the only time these periodic packets include any new information is when counters are over a certain threshold. From this, we observe out last opportunity to delegate functionality: namely, periodically getting switch meta-data and evaluating it against a predefined threshold. We note that delegation does not violate the visibility of the controller provided that the switch generates a report when the packet counters exceed the pre-specific threshold.

Take Aways: We observe that a number of applications run in current SDN networks incur significant

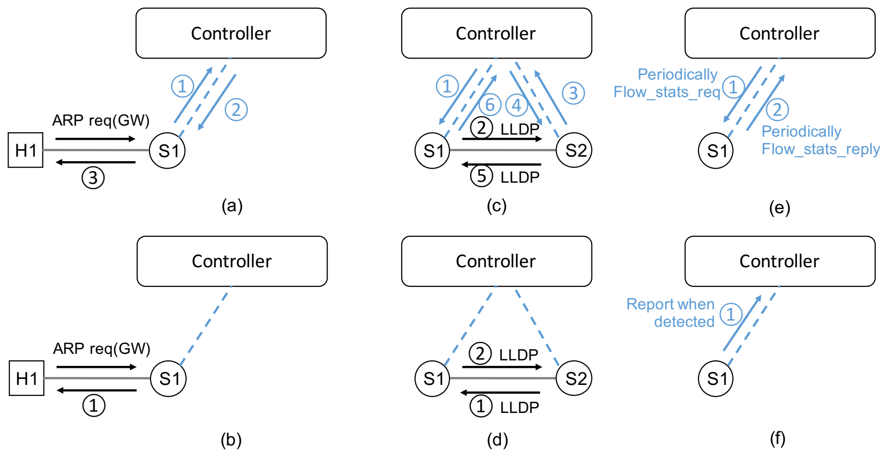


Figure 2: Comparison between OpenFlow (a, c, e) and FOCUS (b, d, f) handling events for ARP, LLDP, and Elephant Flow Detection respectively. The numbers show the sequence of messages exchanged between a controller and switches

overheads due to controller polling the switches or due to devices constantly polling the controller for identical information. Delegating these functionality saves on overheads, bandwidth and CPU, without compromising global control and visibility. Motivated by these observations, we intend to design a set of SDN abstractions, that: (1) allow a controller to offload/delegate responses thus eliminating the need for switches to poll controllers for redundant information; and (2) allows the switches to generate reports to the controller, thus eliminating the need for controller to periodically poll the switches.

2.3 Design Goals

Our observations are similar to those made by prior works [9–11, 31]. These problems persist because prior works [9–11, 31] propose a solution that explores interesting points in the design space that render them hard to adopt: namely, they require new HW primitives. We argue that a practical and deployable design must satisfy the following constraints:

1. **Global Visibility:** Delegating control plane function introduces philosophical concerns: namely, delegation can minimize global visibility and reduce the efficiency of centralized control. Thus, we argue that the API should be designed such that the controller has identical visibility in our environment as it would in a traditional SDN environment.

2. **Local Decisions:** Many of the functions we aim to delegate to the switches often require communication between several switches. For example, LLDP requires two switches to exchange LLDP packets and agree on the status of the switch. Naively designed APIs will introduce complexity and undermine the architecture. Instead, we argue that our abstraction should be designed to require and act solely on local information.

3. **No Hardware Modification:** We argue that rather than burdening the hardware, solutions should be implemented using switch software. Moreover, given the rise of commodities White-Box switches that run Linux [4], we believe that developing a solution that runs on switch CPU will dovetail with orthogonal efforts in industry.

3 Design

FOCUS is an SDN-specific offloading framework that embodies the design goals presented in the last section. FOCUS enables the controller to delegate certain local and stable control-plane functionality to the switch without loss of visibility or control. To support delegation, FOCUS exposes a narrow but expressive API that extends on the traditional OpenFlow data-plane API and thus supports a large number of SDN applications.

The FOCUS API encapsulates each piece of functionality to be offloaded as an FOCUS-rule (Figure 3): a FOCUS-rule consists of two components. The triggers, similar to OpenFlow's match, allows the develop to specify the conditions under which the offloaded function should be applied to control plane events. The action list, orthogonal to OpenFlow's action capabilities, specifies the set of actions to perform when an event matches the pre-specified trigger. For example, for ARP (Table 2), the FOCUS-rule is triggered when a packet matching the ARP-Request is received by the FOCUS-agent. For this

Trigger	Action List	Timeout

Figure 3: FOCUS rule description

packet, the FOCUS-agent creates an ARP request by apply the actions specified in the action list: namely, copying a template of an ARP-reply and filling the template with fields from the original ARP-request and sending this ARP-request out the port it was received. To support this API, FOCUS requires a redesign of the traditional SDN architecture. The redesigned architecture presented in Figure 5, differs from a traditional SDN environment in the following ways:

- **FOCUS-extension:** the controller includes a FOCUS-extension that allows the SDN applications to leverage the FOCUS-API– we call SDN applications using the FOCUS API FOCUS-enabled applications.
- **FOCUS-agent:** at least one switch in the network runs a FOCUS-agent. The FOCUS-agent runs on the Switch CPU alongside the OpenFlow-agent. The FOCUS-agent is charged with setting up triggers and implementing the appropriate actions for each trigger. Further, the FOCUS-agent sits between the OpenFlow-agent and the switch OS, thus allowing it to intercept OF-events and perform offloaded actions before the OpenFlow agent can process the event.

The result of the FOCUS architecture is that fewer control packets are exchanged between switches and the controller which results in CPU savings both on the controller and surprisingly on the switches (Discussed in Section 6). To illustrate these differences, In Figure 4, we present a brief comparison of FOCUS with a traditional SDN deployment: we notice that all control messages need to go to the controller where-as within FOCUS a large number of messages are handled locally by the FOCUS agent running on the switch.

We note two interesting features about this architecture. The architecture for FOCUS is backwards compatible: unmodified applications can run along side modified FOCUS-enabled applications with minimal performance degradation (we discuss this in Section 3.3). Further, FOCUS is incrementally deployable because each offloaded function is local to each switch and thus an application can simultaneously employ FOCUS on FOCUS-enabled switches and interact in a traditional manner with traditional switches. Next we elaborate on the API for configuring FOCUS-rules, then discuss challenges in designing the FOCUS architecture. These range from: (1) ensuring performance and security isolation between the FOCUS

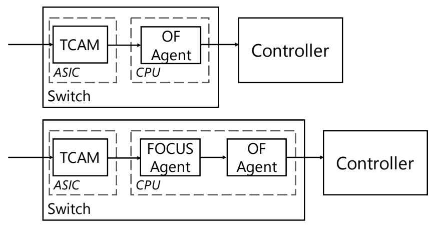


Figure 4: Comparison between FOCUS and traditional SDN deployment

agent and traditional OpenFlow agent (Section 3.3); to (2) ensuring no loss in visibility or control (Section 3.2).

3.1 FOCUS API for expressing FOCUS-rules

A FOCUS-rule, presented in Figure 3, is composed of a *Trigger* and an *Actionlist*, similar in principle to OpenFlow's match and action primitive.

Triggers: Our trigger primitives subsume traditional OpenFlow match primitives and includes timer-based triggers in addition to the ability to match packet fields. Each FOCUS-rule can be triggered by either, a TimeOut or recipient of a packet who fields matches a pre-specified predicate. Next, we elaborate on each trigger.

- **Timer-based:** A timer based trigger enables FOCUS to support periodic functions. For example, constantly polling a switch's resources to determine if certain conditions are met (e.g. Elephant Flow detection) or constantly sending heart-beat messages to other devices in the network (e.g. LLDP). The resolution of the timer is dependent on the hardware characteristics and the Switch-OS: ideally timer resolutions are empirically defined to minimize CPU overheads.
- **Predicate (Packet-matching):** In addition to timers, FOCUS-rules may be triggered by the receipt of packets that match certain conditions. Recall, FOCUS-agent is inserted between the OpenFlow-agent and the switch OS, thus allowing the FOCUS-agent to receive all packets that the dataplane sends to software. FOCUS supports a more expressive set of predicates than OpenFlow. Whereas OpenFlow V1.5 [3] supports 44 predicates in its match primitives, FOCUS utilizes the Type-Length-Value (TLV) scheme [22] which allows FOCUS to support an arbitrary number of predicates. Recall, TLV allows the controller to map the bits in a packet to arbitrary 'types' and then define predicates based on these types.

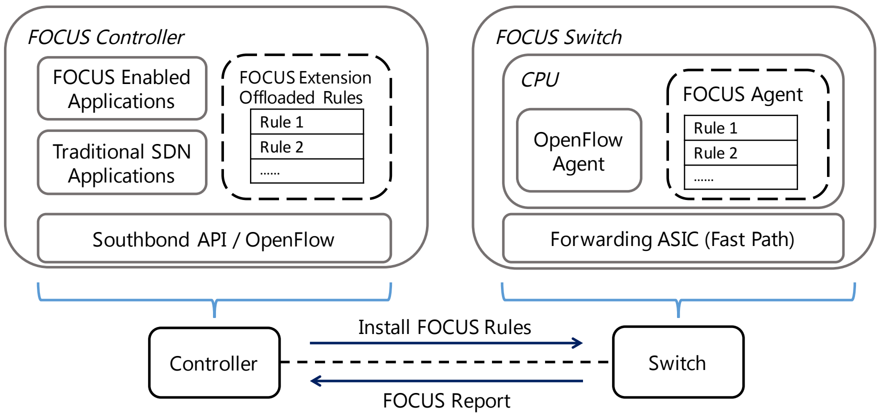


Figure 5: The FOCUS architecture

API Name	Description
Packet Operations	
pkt_compose(t)	Generate an output packet template with type t.
get_field(f)	Read a certain part of an incoming packet.
set_field(f,v)	Write a certain type value v to the output packet.
cksum(t)	Generate a checksum for certain part of the output packet with type t.
pkt_output(p)	Send the output packet to the datapath via port p.
Message Operations	
msg_compose(t)	Generate an empty FOCUS message with type t.
msg_set(f,v)	Add a value v to the message content at field f.
msg_send	Send the FOCUS message to controller.
Function Operations	
rate_with_thresh ($wildcard, flag, thresh$)	Fetch the flow rates and compare with a given threshold. Only the flows with rates larger than threshold will continue to the following commands.

Table 1: FOCUS Primitive APIs

Action-List: FOCUS supports a radically different set of actions than the traditional OpenFlow primitives; the differences in actions underscore the fact that OpenFlow's actions are optimized to support data-plane functionality, whereas FOCUS's actions are optimized to enable delegation of control plane functionality to the switches. At its core, FOCUS supports three groups of actions (Presented in Table 1):

1. **Packet operations** these operations are generally used in conjunction with the predicate-trigger and allow FOCUS to access fields of the input packet, and generate an output packet.

2. **Flow entry operation** this action can be used with either trigger and allows FOCUS to access the meta-data associated with flowtable entries within the datapath. Currently, we support a single action: *rate*$(\cdot,\cdot)$, which examines packet-counters and returns information based on pre-specified threshold.

3. **Message operations** This action is used to manage and deliver the output packets created by the previously described actions.

The primitive APIs form a series of operations and every operation can get the return value of its direct previous operation, as well as the global static fields. A generated packet template waiting to be filled is a typical example of global static fields: it can be touched by a series of operations to form a complete output packet. If a value *EOF* is returned, the operation series will terminate directly, which means the previous operation meets errors or already wraps up the whole process.

3.2 Control and Visibility

A key benefit of SDN is global visibility and centralized control. FOCUS maintains this visibility and central control by limiting the functionality that be delegated to the set of control plane functionality that require information local to the switch and that requires the switch to make no independent decisions outside of that prespecified by the controller (we refer to these function as being stable). When conditions change at the switch, FOCUS provides reports and time-out values that enables the FOCUS agent on the switch to alert the controller of changes.

FOCUS Timeouts Each FOCUS-rule has a time-out associated with it: the time-out expires if the rule is not used within a pre-specified amount of time. Using this information, the FOCUS-agent can eliminate stale FOCUS-rules and inform the application of changes in network conditions. For example, if a switch stops receiving packets that match LLDP, the FOCUS-agent can inform the controller through the FOCUS-extension of a change in network conditions – the controller can treat this as a link failure and react accordingly.

FOCUS Reports The FOCUS-rules and actions are defined to use flows or ports local to a switch. When the status of these ports or flows changes, e.g. port down or flow removed, the FOCUS-agent informs the controller and thus allows the controller to react appropriately.

To support these reports, FOCUS includes special control messages that enables the FOCUS-agent on the switch to update the information at the FOCUS-extension and thus the controller.

3.3 Isolation and Resource Contention

A key challenge in running a FOCUS-agent on the switch lies in its contention for resources with the traditional OpenFlow agent. Fortunately, many switches employ traditional Linux OS, e.g. Cumulus employs Debian [4], thus allowing us to leverage traditional OS isolation and containment techniques. In our architecture, the FOCUS-agent runs as an independent process, communicating with the OpenFlow agent and the controller through two independent TCP connections. This design choice, allows us to reuse process level techniques provided by traditional Operating systems.

4 FOCUS Examples

In this section, we use concrete examples to show how a controller can use the simple set of FOCUS APIs (described in Table 1) to delegate certain functions to switches. When a controller delegates a function to switches, it reduces both its computational overhead and its communication overhead with the switches.

At a high-level, a FOCUS controller can delegate two types of functions to switches: 1) packet-match triggered functions, and 2) periodic timer triggered functions. We describe each in turn.

4.1 Packet-Match Triggered Functions

Many response/reply protocols fall into this category. In this type of functions, an incoming packet that matches certain patterns triggers an outgoing packet. If the function is local and stable, i.e., the same matched patterns trigger the same outgoing packets using switch-local information, a controller can delegate this function to switches.

We show a concrete example using ARP. Other examples include the ICMP Time Exceeded function, the MAC address learning function, and the DHCP relay function.

ARP Reply for Default Gateway: Recall that we describe in § 2, without FOCUS, an SDN controller must reply an ARP request to the default gateway from every host. With FOCUS, a controller can offload this function to a switch by installing a new FOCUS rule and a set of action items. It can either proactively offload this function at all edge switches or reactively install this information at an edge switch after it receives the first OF_Packet_In packet triggered by an ARP query received by an edge switch.

We show in Table 2 how to implement this function using the FOCUS API. A controller installs a packet match rule in a switch's FOCUS rule table. If a packet matches the type "ARP" and the ARP query is for the default gateway IP, then the FOCUS agent will execute the installed actions. The first action is to generate an ARP reply packet template using the `pkt_compose()` function. Then a seqence of `get_field()` and `set_field()` actions set the corresponding fields in the outgoing packet from the incoming packet. For example, the destination MAC address in the outgoing packet is set from the source MAC address of the incoming packet. After the corresponding fields in the packet are set, the last action sends out the outgoing ARP packet.

In addition to ARP replies to the default gateway address, it is possible to offload ARP replies to all host IP addresses to switches as done in [10]. It is a design decision an SDN operator can make. We do not show the examples here for clarity.

Interested readers can find more examples of packet-match triggered functions such as **ICMP Time Exceeded** and **DHCP Relay** in Appendix A.

Trigger	Actions
ARP target_IP=GW_IP	*pkt_compose*(ARP)
	get_field(src_MAC)
	set_field(dst_MAC, ret[1])
	set_field(target_MAC, ret)
	get_field(src_IP)
	set_field(target_IP, ret)
	pkt_output(in_port)

[1] Return value of the last operation.

Table 2: **How a controller uses the FOCUS API to delegate the ARP reply for the default gateway function.**

4.2 Periodic Timer-Triggered Functions

Another category of functions a controller can delegate to a switch is periodic timer triggered functions. To do so, a controller installs a FOCUS rule that has a time and a set of associated action items in a switch's FOCUS rule table. When the timer times out, the FOCUS executes the installed action items.

We show two examples here. One is the periodic LLDP link discovery function, and the other is the elephant flow detection function.

LLDP: In a traditional SDN deployment, a controller periodically directs a switch to send and receive LLDP messages for each port the swtich attaches to as shown in Figure 2. With FOCUS, a controller can offload the periodic sending and receipt of an LLDP message to a switch. After the controller generates the unique LLDP packet for a switch's port, and learns the existence of the physical link, it can install a periodic LLDP packet generation rule at the switch and directs it to send the packet periodically through the port. The rule includes two actions: an action to generate the LLDP packet and an action to send it along a specified port, as shown in Table 3. After the timer times out periodically, the switch sends the LLDP packet out along the specified port, without sending any packet to the controller.

A controller also needs to install a receiver side function (not shown for simplicity) to process the received LLDP packet. The receiver function includes a timer and an LLDP packet matching trigger, but has no action item. If a switch receives an LLDP packet from another switch, the FOCUS agent will detect a packet match event, and cancel the timer; otherwise, the timer times out, indictaing a link failure has occured. The FOCUS agent will send a control message OF_Packet_In to the controller.

Elephant Flow Detection: A controller can use the threshold rate detection function provided in the FOCUS

Trigger	Actions
Timer	*pkt_compose*(LLDP)
Trigger	*pkt_output*(PORT_NO)

Table 3: **How a controller delegates a periodic LLDP link discovery packet to switches.**

API to delegate elephant flow detection function as well as other packet counter related functions such as load inbalancing detection or DDoS flow detection to switches. Without FOCUS, a controller must send periodic packets to poll the packet counters, even if no flows are detected.

Table 4 shows how a controller can use the FOCUS API to implement the elephant flow detection function. The controller installs a periodic timer and uses the `rate_with_thresh()` API to detect elephant flows. The first parameter of `rate_with_thresh()` specifies the flow matching descriptors. The third parameter specifies the flow rate threshold. The second specifies whether all individual flows whose rates exceed the threshold should be returned (True) or an aggregate flow should be returned (False).

When the timer times out, the FOCUS agent first executes the `rate_with_thresh()` action. The pseudocode to implement this API is shown in Algorithm 1. The agent reads the packet counters whose flows match the specified flow descriptors, and returns all the matched flows whose packet rates exceed the controller specified threshold together with their packet rates.

Trigger	Actions
Timer Trigger	*rate_with_thresh* ((*,*,*,*,*)[2], *True*, thresh[3])
	msg_compose(ELEPHANT)
	msg_set(FLOW_DESCRIPTOR, flow_descs[4])
	msg_set(PKT_RATE, counters[4])
	msg_send

[2] A five tuple (srcIP, dstIP, srcPort, dstPort, proto) specifies which flows' packet counters to monitor. In this example, all flows are monitored.
[3] A flow rate threshold defined by controller.
[4] The packet counter values returned from *rate_with_thresh*. A controller can infer a flow's packet rate from a packet counter value.

Table 4: **How a controller can delegate the elephant flow detection function to switches.**

We note that by varying the flow descriptor parameter in the `rate_with_thresh` API, a controller can delegate at least two other functions to switches. One is DDoS attack detection. Here, an controller specifies the monitored server IP as the first parameter in the `rate_with_thresh` function: `(*,dstIP, *,*,*)`. The second parameter is set as false. When the aggregate traffic reaching the destination IP exceeds a threshold, a controller receives a DoS attack notification from the switches.

The second example is the load balancing function. Here the controller specifies the monitored server, port, and protocol in the first parameter of the `rate_with_thresh` function: `(*, dstIP, *, dstPort, proto)`, and set the second parameter to true. All individual flows whose rate exceed the specified threshold will be returned to the controller's load balancing application, and the controller can decide how to reroute those flows to achieve load balancing.

Algorithm 1 The *rate_with_thresh* API

```
procedure RATE_WITH_THRESH(flow_desc,flag,thresh)
    cur_counters = read_counters(flow_desc)
    if flag then
        diffs = cur_counters - prev_counters
        prev_counters = cur_counters
        for d in diffs do
            if $d > thresh$ then
                result.append(d)
        if result.length() == 0 then
            return EOF
        else
            flow_descs = result.flow_descs
            counters = result.counters
    else
        cur = sum(counters)
        diff = cur - prev
        prev = cur
        if $diff < thresh$ then
            return EOF
        else
            flow_desc.append(flow_desc)
            counters.append(diff)
```

5 Implementation

We implemented our prototype of FOCUS in 1000 lines of Java code (for the Controller)and 700 lines of C code (for the switch). To do this, we modified the Floodlight [1] controller to support the FOCUS interface and extensions and modified Open vSwitch [2] to export the FOCUS interface. Next, we discuss the implementations highlights for controller, switch, and applications changes.

Controller Changes: To implement the FOCUS extension, we added a new class, that exposes the FOCUS interface to applications. To efficiently manage offloaded functions, the FOCUS extension maintains a hashtable called "Offloaded Function List". In response to a request to install/update FOCUS-rules in a switch, the FOCUS extension updates its internal structures and sends an Experimental message to the switch containing the type of function to offload and the set of actions.

This class subscribes to two OpenFlow events. The first, Switch Hello messages, this event is used to determine the liveliness of switches and the validity of entries in the "Offloaded Function List" table. When the failure of a Hello message indicates that a switch is offline, the class notifies the appropriate application of invalid FOCUS-rules. The second, OpenFlow Experimental message, this event is delivered to the application for processing – FOCUS uses the "Offloaded Function List" to determine which App to send the experimental message to.

Switch Changes: At the switch, we alter the processing pipeline on the switch OS to send all packets from the data-plane to FOCUS agent, before the OpenFlow agent. The FOCUS agent implements the offload table, described in § 3, as a set of two parts: trigger table and extended action table.

Although our current implementation of FOCUS agent runs in Open vSwitch, given the recent push for a commoditizing OS for SDN switches. We believe that with minor modifications our FOCUS agent can run arbitrary SDN hardware switches, such as, Facebook's FBOSS, DELL S6000-ON, HP Altoline 5712.

Application Changes: We re-wrote Floodlight controller applications to leverage the new functionality provided by FOCUS. In Section 6, we examine the performance differences for three of these applications. We present the FOCUS-rules for these applications in Section 4 and Appendix A.

6 Evaluation

In evaluating FOCUS, we aim to answer the following questions: What are the benefits of employing FOCUS? What are the costs of offloading functionality to the switches? How do the benefits and overheads of FOCUS vary across different the different API calls (and applications)?

6.1 Evaluation Setup

We evaluate FOCUS using the Mininet [18] emulator. All our experiments are run on a Dell R620 server with 8G RAM and 2.80GHz Quad Core Intel CPU running

Ubuntu 14.04 LTS. Recall, we developed our applications on the Floodlight controller, thus we will be evaluation performance of the Floodlight controller with and without FOCUS extensions.

Topology We evaluate FOCUS on three types of topology: a simple topology for the ARP, a complex mesh-like topology for the LLDP, and a linear topology for the elephant flow detection.

Metrics To understand the costs and benefits of applying FOCUS, we analyze the CPU utilization (for controller and switch) and the total number of control messages exchanged (this serves as a proxy for control plane bandwidth). In all situations, we take especial care to ensure that the controller is only running the application we are evaluating.

6.2 ARP

We begin, in Figure 6(a), by analyzing the impact of FOCUS on the ARP applications. Specifically, the controller's and switch's CPU utilization as a function of the number of ARP requests generated. We observe that without FOCUS, the controller's CPU utilization is a linear function of the number of ARP request whereas with FOCUS, the CPU utilization at the controller is constant. As expected, the CPU saving is nearly 100% because the switch takes over all the tasks. As for switch's CPU utilization, given that FOCUS minimizes controller's CPU utilization by overloading to the switch, intuitively, we expect to see an increase in switch CPU utilization. Interestingly, our experiments show the exact opposite – we observe that FOCUS also reduces CPU utilization at the switches. To understand this phenomenon, we analyze the OpenFlow agents (OS processes) running the switches and observe that FOCUS eliminates the generation of additional Packet-In to the controller which results in a significant savings. To confirm this, in Figure 7(a) we present the number of control messages generated as a function of packets sent. From this figure it is apparent that FOCUS virtually eliminates all communicate between the controller and switches, thus eliminating the overheads for generating and processing messages to/from the controller.

Finally, in Figure 7(b), we compare the ARP resolution times. We observe that by offloading to the switches and eliminating the need to involve the controller in ARP resolution, FOCUS improves ARP response times by more than 20 ms.

6.3 LLDP

Next, in Figure 6(b), we examine the performance of the LLDP application. We observe, similar trends in the LLDP experiments as with the ARP experiments with one difference: the CPU savings increase after 4000 links. This increase occurs because in the Mininet simulator, due to processor sharing, each switch incurs significantly more overhead after the topology has over 4000 links – whereas with FOCUS this overhead is not incurred.

Finally, we analyze the traffic load for LLDP in Figure 7(c). We observe that FOCUS reduces control plane traffic by 50% on average – this is because the controller no longer needs to poll the switches, the switches actively report changes to the topology. However, unlike ARP, in LLDP there is still a significant amount of control traffic even with FOCUS because in ARP the switch no longer needs to communicate with the Controller, whereas in LLDP the switches still need to periodically report changes to the controller.

6.4 Elephant Flow Detection

Our last application implements Elephant flow detection, unlike the previous two applications, this application employs the `rate_with_thresh()` API. In Figure 6(c), we present the results of our experiment, from which we can find that FOCUS similarly decreases CPU utilization on both the switch and the controller for similar reasons as with ARP – with more flows the CPU on the switch does more work. We also analyzed network traffic, omitted here due to space, and found its pattern to be similar to the LLDP.

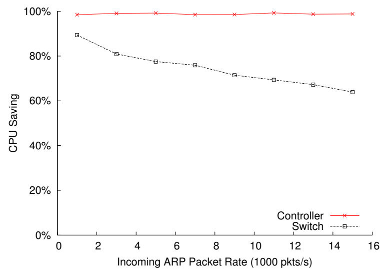
6.5 Memory Limitations

In OpenFlow switches, hardware and software, Flowtable entries and memory are crucial. Motivated by this fact, next, we examine the memory overheads of FOCUS. In our experiment, we observed that FOCUS scaled linearly with the number of actions and that FOCUS never required more than 4MBs of memory.

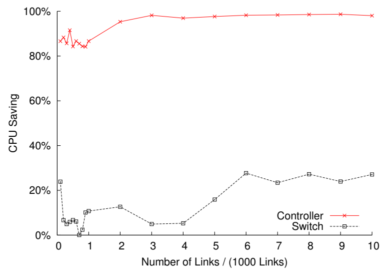
7 Related Work

The most closely related works to FOCUS aim to scale the control plane by either introducing novel switch primitives [11, 20, 31, 34] or by distributing the control plane [13, 14, 16, 17, 28, 30]. Whereas distributed controller architectures partition visibility of network dynamics and control of the network among controllers, in FOCUS, the controller retains visibility and total control.

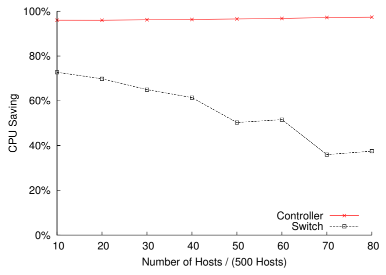
Unlike existing techniques on novel switch primitives [11,20,31,34], FOCUS only delegates control plane functions to a switch's software stack, and does not require changes to the switch's hardware. We made this design decision for simplicity of design and ease of deployment. This difference makes FOCUS immediately



(a) ARP

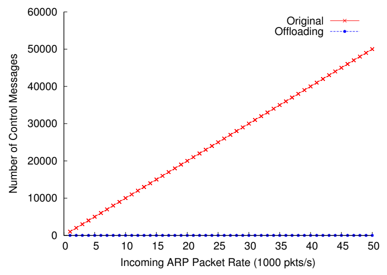


(b) LLDP

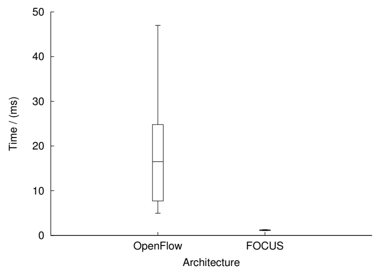


(c) Elephant Flow Detection

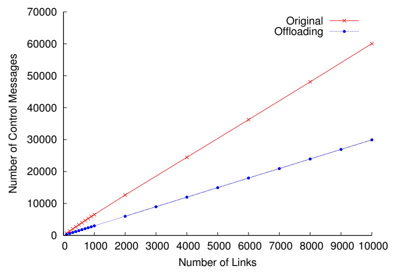
Figure 6: The CPU saving for different SDN applications: (a) ARP, (B) LLDP, and (C) Elephant Flow Detection.



(a) ARP Packet Traffic



(b) ARP Response Time



(c) LLDP Packet Traffic

Figure 7: Results Other than CPU Utilization

applicable to all Whitebox SDN switches. Further, FOCUS supports delegation of richer set of functionality than exist approaches [10, 11].

Identification of stable functionality is in principle similar to the concepts of local functions identified in prior works [14, 17, 23]. Unlike orthogonal approaches [14, 17, 23], which define locality based on a set of switches, FOCUS focuses on locality relative to one switch and network configuration.

The introduction of a programming interface to support delegation of local functions, while similar to the interfaces provided by Kandoo [14], is fundamentally different. FOCUS limits the developer to a narrow and simple interface to ensure simplicity at the switch, whereas Kandoo [14] allows developers to develop arbitrarily rich offload messages due to the complexity supported by the local controllers.

Other related works [15, 19, 29, 30] propose higher level programming abstractions that simplify development and minimize errors. As part of future work, we intend to explore such high level programming languages [15, 19, 29, 30] and determine how these languages can enable automated detection and offloading of stable functions.

8 Conclusion

In this paper, we study how to improve the scalability of an SDN controller by enabling a controller to delegate some control functions to switches. We study the tradeoffs between the flexibility of delegation and the cost of delegation. We choose a design point where a controller can use a simple set of APIs to delegate a variety of functions, which we call stable local functions. These functions remain stable over time as long as the network configuration does not change and they only require input local to a switch to compute. We describe the FOCUS APIs and use concrete examples (including ARP replies, LLDP, and elephant flow detection) to show how a controller can use these simple APIs to delegate packet-match triggered functions and periodic timer triggered functions.

Finally, we implement ARP replies, LLDP, and elephant flow detection functions using the FOCUS API, and show that the FOCUS design can significantly reduce a controller's as well as a switch's communication and computational overhead. In our experiments, FOCUS can reduce a controller's communication overhead by 50% to nearly 100%, and the computational overhead by 80% to 98%. Because FOCUS reduces the communication overhead between a switch and a controller, it can reduce a switch's overall computational overhead by

60% to 90% for ARP replies, around 35% for large-scaled LLDP and 40% to 80% for elephant flow detection, even though it adds additional functions to a switch.

Appendices

A More FOCUS Examples

A.1 ICMP Time Exceeded

Moreover, we also discuss more API implementation examples for a wider range of functions. TTL expiration can be widely used in traceroute. FOCUS can also be responsible to make up ICMP message for TTL expiration. An ICMP packet TTL value 0 triggers the actions in Table 5.

Firstly, a packet template for ICMP reply is made by API calling. The first 28 bytes of the original packet will be filled in the template to form a new packet. Because this new ICMP packet needs to be sent out via data plane, checksums for both L3 header and ICMP header is required. After the packet is complete, the switch could sent it back to the incoming port.

Trigger	Actions
ICMP TTL=0	*pkt_compose*(ICMP)
	get_field(L3_Head, offset=0, len=28)
	set_field(L3_Content, offset=4)
	cksum(L3)
	cksum(ICMP)
	pkt_output(in_port)

Table 5: Operation Series for ICMP Time Exceeded

A.2 DHCP Relay

Table 6 shows the implementation of DHCP relay. When DHCP discover and DHCP request packets are received by the FOCUS switch, the triggers will fire if the DHCP packet's ingress physical port matches one of the rules. According to the the mapping between the physical ingress ports and subnets, FOCUS will first generate a DHCP packet, and copy the DHCP content after the DHCP relay agent field from the received packet, then send out the packet to the corresponding egress physical port. When receiving a DHCP packet from the DHCP server, FOCUS switch will match the rules based on dst_IP. Once the trigger is fired, a predefined DHCP packet is generated. Then the DHCP content will be copied from the original packet. Finally the generated packet will be send to certain physical ports belongs to the corresponding subnet.

Trigger	Actions
L2_Broadcast In_Port DHCP	*pkt_compose*(DHCP)
	get_field(L4_Content, offset=144, len=-1[5])
	set_field(L4_Content, offset=144)
	pkt_output(dhcp_srv_port)
DHCP_Srv_Port DHCP_GW_IP	*pkt_compose*(DHCP)
	get_field(L4_Content, offset=0, len=-1)
	set_field(L4_Content, offset=0)
	pkt_output(dhcp_client_port)

[5] "len=-1" means to the end of packet.

Table 6: Operation Series for DHCP Relay

A.3 DDoS Attack Detection

Similar to elephant flow detection, another application based on packet counter is DDoS Attack Detection. Table 7 shows the action list consists of FOCUS APIs for this application. Different from the previous example, the second parameter of $rate\_with\_thresh(\cdot,\cdot,\cdot)$ is set to *True* because DDoS attack detection aims to find the aggregate flow size for a category of flows to determine whether it exceeds a reasonable value.

A.4 Load Balancer

We also propose Load Balancer as another example for packet counter. Table 8 shows the action list for load balancer.

B Theoretical Analysis

In this section, we provide the theoretical result to show the saving introduced by FOCUS. The results are listed at Table 9.

Trigger	Actions
Timer Trigger	*rate_with_thresh* (wildcard[6], *True*, thresh)
	msg_compose(DDOS)
	msg_set(WILDCARD, wildcards)
	msg_set(PKT_RATE, counters)
	msg_send

[6] Wildcard for the a specific source.

Table 7: Operation Series for DDoS Attack Detection

Trigger	Actions
Timer Trigger	*rate_with_thresh* (wildcard[7], *False*, thresh)
	msg_compose(LB)
	msg_set(WILDCARD, wildcards)
	msg_set(PKT_RATE, counters)
	msg_send

[7] Wildcard for all flows.

Table 8: Operation Series for Load Balancer

B.1 ARP Reply for Default Gateway

With FOCUS, the switch can take the responsibility to reply the ARP request for default gateway. If we use h to denote the total number of hosts in an SDN network, R the rate at which an ARP query for a default gateway is sent in the network, s_e the total number of edge switches in the network, and $\bar{h}$ the average number of hosts an edge switch is connected to, then in a traditional SDN network, the total number of messages a controller sends and receives to answer an ARP query for the default gateway is $2h*R$, and the computational overhead a controller spends on processing those messages scales as $O(h*R)$. In contrast, with FOCUS, a controller sends and receives at most $2s_e$ messages (for passive trigger mode, while it is s_e for active trigger mode) in response to ARP queries for the default gateway, and the computational overhead scales with the number of edge switches in the network $O(s_e)$. This is result is shown in Table 9 in the row starts with "ARP GW". As we can see in § 6, when a network has many hosts, FOCUS significantly reduces a controller's the computational and communication load.

B.2 LLDP

After the switches are able to send LLDP packet periodically, the controller no longer needs to take care of such repetitive if nothing happens (e.g., link failure). If we use p to denote the total number of switch ports in an SDN network, and R the rate at which a controller sends an LLDP packet, we can see that without FOCUS, the number of LLDP messages a controller sends and receives is $2p*R$, and its LLDP message processing overhead scales as $O(p*R)$. In contrast, with FOCUS, the number of LLDP messages a controller sends and receives is p, and its LLDP message processing overhead is $O(p)$, if we assume the network is stable and does not suffer frequent link flapping.

B.3 Elephant Flow Detection

With FOCUS, the switches can set a threshold for flow rate and only an exceeding event will trigger an OF_Packet_In to the controller, which means when the flow rate is not significant enough, such flow will not occupy controller resource. By setting the threshold properly, the controller can still acquire the information it needs without bother to collect a bunch of information for the small flows.

If we use h to denote the number of hosts in the network, and R the rate at which controller checks the flow rate within switch, then without FOCUS, a controller will receive $h*R$ messages from the switch to report the flow status. And the computational overhead is therefore $O(h*R)$.

On the other hand, with FOCUS, the controller will only receive $h*\bar{f}_e$ messages from the switch, where $\bar{f}_e$ is the average number of elephant flows for each switch. Similarly, the computational overhead turns to $O(h*\bar{f}_e)$ as well.

B.4 Learning Switch

Compared to send an OF_Packet_In to the controller for reporting, if the SDN network employs FOCUS, the controller can cache the mapping between a host's MAC address to its incoming port at an edge switch. The switch sends a new message to the controller only when it receives an ARP message with either a new source MAC address or a new incoming port. To deal with host mobility, an controller can set an expiration time to each entry in the MAC address table, both at its centralized MAC table and at the cached entry in each switch. After a host

Protocol	Resource			
	# of Control Messages		Controller CPU	
	No Offloading	Offloading	No Offloading	Offloading
Arping GW	$2h*R$	s_e	$O(h*R)$	$O(s_e)$
LLDP	$2p*R$	p	$O(p*R)$	$O(p)$
Elephant Flow Detection	$h*R$	$h*\bar{f_e}$	$O(h*R)$	$O(h*\bar{f_e})$
Learning Switch	$h*R$	h/T	$O(h*R)$	$O(h/T)$

Table 9: Overhead in Reactive Controller Compared to Leveraging Offloading Strategy.
s_e: The number of edge switches. h: The number of hosts. p: The number of switch ports involved in the specified protocol; R: The sending rate for the corresponding protocol packets; c_i: A constant; T: Expiration time; f_e: The average number of elephant flows for each switch.

leaves the network or a switch, the entry will expire after the expiration time.

If we use h to denote the number of hosts in the network, and R the rate at which a host sends an ARP packet, then without FOCUS, a controller receives ARP messages at a rate $h*R$, and its computational overhead also scales with $O(h*R)$.

In contrast, with FOCUS, a controller receives ARP messages at a rate h/T, where T is the expiration time of an MAC entry. Correspondingly, the computational load reduces to $O(h/T)$.

References

[1] Floodlight. `http://www.projectfloodlight.org/floodlight/`.

[2] Open vSwitch. `http://openvswitch.org`.

[3] OpenFlow Switch Specification V1.5.0. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.0.noipr.pdf`.

[4] What's Inside Cumulus Linux for Networking? `http://www.enterprisenetworkingplanet.com/netos/whats-inside-cumulus-linux-for-networking.html`.

[5] AL-FARES, M., RADHAKRISHNAN, S., RAGHAVAN, B., HUANG, N., AND VAHDAT, A. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI* (2010), vol. 10, pp. 19–19.

[6] BALLARD, J. R., RAE, I., AND AKELLA, A. Extensible and Scalable Network Monitoring Using OpenSAFE. *Proc. INM/WREN* (2010).

[7] BENSON, T., ANAND, A., AKELLA, A., AND ZHANG, M. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *Proceedings of the Seventh COnference on emerging Networking EXperiments and Technologies* (2011), ACM, p. 8.

[8] BERDE, P., GEROLA, M., HART, J., HIGUCHI, Y., KOBAYASHI, M., KOIDE, T., LANTZ, B., O'CONNOR, B., RADOSLAVOV, P., SNOW, W., ET AL. ONOS: Towards an Open, Distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking* (2014), ACM, pp. 1–6.

[9] BIANCHI, G., BONOLA, M., CAPONE, A., AND CASCONE, C. Openstate: programming platform-independent stateful openflow applications inside the switch. *ACM SIGCOMM Computer Communication Review 44*, 2 (2014), 44–51.

[10] BIFULCON, R., BOITE, J., BOUET, M., AND SCHNEIDER, F. Improveing SDN with InSPired Switches. *ACM SIGCOMM SOSR* (2016).

[11] CURTIS, A. R., MOGUL, J. C., TOURRILHES, J., YALAGANDULA, P., SHARMA, P., AND BANERJEE, S. DevoFlow: Scaling Flow Management for High-Performance Networks. In *ACM SIGCOMM Computer Communication Review* (2011), vol. 41, ACM, pp. 254–265.

[12] DAS, A., LUMEZANU, C., ZHANG, Y., SINGH, V., JIANG, G., AND YU, C. Transparent and Flex-

ible Network Management for Big Data Processing in the Cloud. In *5th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13)* (2013).

[13] DIXIT, A., HAO, F., MUKHERJEE, S., LAKSHMAN, T., AND KOMPELLA, R. Towards an Elastic Distributed SDN Controller. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 7–12.

[14] HASSAS YEGANEH, S., AND GANJALI, Y. Kandoo: A Framework for Efficient and Scalable Offloading of Control Applications. In *Proceedings of the first workshop on Hot topics in software defined networks* (2012), ACM, pp. 19–24.

[15] KIM, H., REICH, J., GUPTA, A., SHAHBAZ, M., FEAMSTER, N., AND CLARK, R. Kinetic: Verifiable Dynamic Network Control.

[16] KOPONEN, T., CASADO, M., GUDE, N., STRIBLING, J., POUTIEVSKI, L., ZHU, M., RAMANATHAN, R., IWATA, Y., INOUE, H., HAMA, T., ET AL. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI* (2010), vol. 10, pp. 1–6.

[17] KRISHNAMURTHY, A., CHANDRABOSE, S. P., AND GEMBER-JACOBSON, A. Pratyaastha: An Efficient Elastic Distributed SDN Control Plane. In *Proceedings of the third workshop on Hot topics in software defined networking* (2014), ACM, pp. 133–138.

[18] LANTZ, B., HELLER, B., AND MCKEOWN, N. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks* (2010), ACM, p. 19.

[19] MONSANTO, C., REICH, J., FOSTER, N., REXFORD, J., WALKER, D., ET AL. Composing Software Defined Networks. In *NSDI* (2013), pp. 1–13.

[20] MOSHREF, M., BHARGAVA, A., GUPTA, A., YU, M., AND GOVINDAN, R. Flow-level State Transition as a New Switch Primitive for SDN. In *Proceedings of the third workshop on Hot topics in software defined networking* (2014), ACM, pp. 61–66.

[21] NICIRA NETWORKS. Open vSwitch, An Open Virtual Switch. *http://openvswitch.org/* (2010).

[22] PRZYGIENDA, T. Reserved Type, Length and Value (TLV) Codepoints in Intermediate System to Intermediate System.

[23] SCHMID, S., AND SUOMELA, J. Exploiting Locality in Distributed SDN Control. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* (2013), ACM, pp. 121–126.

[24] SHIN, S., PORRAS, P. A., YEGNESWARAN, V., FONG, M. W., GU, G., AND TYSON, M. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *NDSS* (2013).

[25] SUH, J., KWON, T. T., DIXON, C., FELTER, W., AND CARTER, J. OpenSample: A Low-Latency, Sampling-Based Measurement Platform for Commodity SDN. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on* (2014), IEEE, pp. 228–237.

[26] TAVAKOLI, A., CASADO, M., KOPONEN, T., AND SHENKER, S. Applying NOX to the datacenter. In *Eight ACM Workshop on Hot Topics in Networks (HotNets-VIII), HOTNETS '09, New York City, NY, USA, October 22-23, 2009* (2009).

[27] TENNENHOUSE, D. L., AND WETHERALL, D. J. Towards an active network architecture. *SIGCOMM Comput. Commun. Rev. 37*, 5 (Oct. 2007), 81–94.

[28] TOOTOONCHIAN, A., AND GANJALI, Y. HyperFlow: A Distributed Control Plane for OpenFlow. In *Proceedings of the 2010 internet network management conference on Research on enterprise networking* (2010), USENIX Association, pp. 3–3.

[29] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: Simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM Computer Communication Review* (2013), vol. 43, ACM, pp. 87–98.

[30] WEDDE, H. F., FAROOQ, M., AND ZHANG, Y. BeeHive: An Efficient Fault-Tolerant Routing Algorithm Inspired by Honey Bee Behavior. In *Ant colony optimization and swarm intelligence*. Springer, 2004, pp. 83–94.

[31] YU, M., REXFORD, J., FREEDMAN, M. J., AND WANG, J. Scalable Flow-Based Networking with DIFANE. *ACM SIGCOMM Computer Communication Review 41*, 4 (2011), 351–362.

[32] ZAALOUK, A., KHONDOKER, R., MARX, R., AND BAYAROU, K. OrchSec: An Orchestrator-Based Architecture for Enhancing Network-Security Using Network Monitoring and SDN Control Functions. In *Network Operations*

and Management Symposium (NOMS), 2014 IEEE (2014), IEEE, pp. 1–9.

[33] ZHOU, Y., RUAN, L., XIAO, L., AND LIU, R. A Method for Load Balancing based on Software Defined Network. *Advanced Science and Technology Letters 45* (2014), 43–48.

[34] ZHU, S., BI, J., SUN, C., WU, C., AND HU, H. SDPA: Enhancing Stateful Forwarding for Software-Defined Networking. In *Network Protocols (ICNP), 2015 IEEE 23nd International Conference on* (2015), IEEE.