Controlling Race Conditions in OpenFlow to Accelerate Application Verification and Packet Forwarding

Xiaoye Steven Sun, Apoorv Agarwal, and T. S. Eugene Ng

***Abstract*—OpenFlow is a Software Defined Networking (SDN) protocol that is being deployed in many network systems. SDN application verification takes an important role in guaranteeing the correctness of the application. Through our investigation, we discover that application verification can be very inefficient under the OpenFlow protocol since there are many race conditions between the data packets and control plane messages. Furthermore, these race conditions also increase the control plane workload and packet forwarding delay. We propose Attendre, an OpenFlow extension, to mitigate the ill effects of the race conditions in OpenFlow networks. We have implemented Attendre in NICE (a model checking verifier), Open vSwitch (a software virtual switch), and NOX (an OpenFlow controller). Experiments show that Attendre can reduce verification time by several orders of magnitude, and significantly reduce TCP connection setup time.**

***Index Terms*—OpenFlow, software-defined network, race condition, model checking, verification, forwarding delay.**

I. Introduction

The OpenFlow protocol enables the programmatic and centralized control of a network's behavior. Software application programs running on a centralized OpenFlow controller can observe virtually any aspects of an OpenFlow-enabled network and correspondingly dictate the behavior of the network by sending command messages to directly configure packet processing rules on the OpenFlow-enabled network switches. This centralized, user-programmable control plane architecture is generally referred to as Software-Defined Networking (SDN). OpenFlow is currently the *de facto* standard protocol for SDN in the industry [1].

One of the most important usages of SDN is in fine-grained network security policy enforcement in enterprise networks to combat the unauthorized resource access threat [2]–[7]. To realize fine-grained network security policies, a critical feature supported by OpenFlow is the so-called packet-in messages. Using this feature, a traffic flow that is not known to the network switches is directed, via packet-in messages, to the centralized controller for processing. There, an application program can reactively make sophisticated decisions about how this traffic flow should be handled (e.g. authorized and routed, denied and filtered, redirected, monitored, logged etc.) based on a user customizable policy.

In such usages of SDN, the correctness of the SDN controller application programs is paramount. The SDN controller applications are responsible for both realizing the security policy and for recovering the network from failures [8] in switches, load balancers, etc. [9]. It is therefore critical to rigorously verify the correctness of SDN controller applications.

Among all automated SDN verification approaches proposed (a broader discussion of the various approaches can be found in Section IX), model checking is the most powerful one, since it is capable of detecting the widest range of misbehavior (e.g., forwarding loops, reachability failures, policy violations, lack of liveness) in SDN applications even before these applications are deployed. In model checking, the entire network, including the switches, the hosts and the controller running SDN applications, is modeled as a state machine. The verifier explores the entire state space and checks for network property invariant violations by simulating every possible event ordering inside the entire network.

Unfortunately, controlling the network in a reactive way using the packet-in message makes model checking based application verification highly inefficient. The underlying reason is that the data packets and the OpenFlow messages exhibit many race conditions, and each race condition exacerbates the number of state transitions that the model checker needs to explore. We further discover that the race condition also increases packet forwarding delay and the processing overhead on the switches and the controller. These problems greatly limit the practicality of SDN application verification as a means to ensure network correctness, and lower the performance of an SDN network unnecessarily.

This paper is the first study to systematically analyze the origins and impacts of such race conditions in the OpenFlow protocol and to explore new protocol designs to mitigate the problems they cause. We make the following contributions[1]:

1) Identify Race Conditions in an OpenFlow Network: We present and illustrate the following three types of race

Manuscript received April 21, 2014; revised October 27, 2014, February 6, 2015, and March 15, 2015; accepted March 29, 2015. Date of publication April 6, 2015; date of current version June 12, 2015. This research was sponsored by the NSF under CNS-1422925, CNS-1305379, and CNS-1162270, by an Alfred P. Sloan Research Fellowship, an IBM Faculty Award, and by Microsoft Corp. The associate editor coordinating the review of this paper and approving it for publication was O. Festor.

X. S. Sun is with the Department of Electrical and Computer Engineering, Rice University, Houston, TX 77005 USA (e-mail: xiaoye.sun@rice.edu).

A. Agarwal and T. S. Eugene Ng are with the Department of Computer Science, Rice University, Houston, TX 77005 USA.

Digital Object Identifier 10.1109/TNSM.2015.2419975

[1]The following discussion assumes the reader has a basic understanding of the OpenFlow protocol and its terminologies. Section II gives a brief tutorial.

conditions in detail with two different network scenarios.[2] The first two types of races exist between a data packet and a command message. The third type of race exists between two command messages.

- At the switch sending out a packet-in message to the controller, the following packets in the same flow of the packet in the packet-in race against the command messages sent to the switch for processing this flow.
- When the controller receives a packet-in and decides to send flow-mod messages to multiple switches for the data flow, at the switches not sending the packet-in but receiving the flow-mods, the data packets in the flow race against the flow-mods.
- Command messages sent to the same switch for the same flow race against each other.

2) Describe the Ill Effects of the Race Conditions: Using state transition diagrams, we point out the ill effects of the races in model checking based OpenFlow application verification. We also describe other ill effects, like increased packet forwarding delay and increased workload on the switches and the controller.

3) Control and Mitigate the Ill Effects of Race Conditions With Attendre: We present Attendre, an extension of the OpenFlow protocol to control the race conditions so as to mitigate the ill effects of races. By controlling these races, we mean that the messaging behavior of a switch is as if the outcomes of the races are deterministic. By controlling the outcomes of the race conditions, Attendre eliminates many unnecessary network states as well as many unnecessary control plane messages, which are the root causes for inefficient verification and increased forwarding delay. We present the changes to the OpenFlow protocol and the changes to switches in detail.

4) Implement Attendre and Quantify the Benefits of Attendre: We have implemented Attendre in NICE [10] (a model checking based OpenFlow application verifier), in Open vSwitch [11] (a software OpenFlow switch) and in NOX [12] (an OpenFlow controller platform). With these implementations, we demonstrate that compared to the original OpenFlow protocol, Attendre reduces the verification execution time by several orders of magnitude; we also show that Attendre greatly reduces the TCP connection setup time. To theoretically understand the forwarding delay in an OpenFlow network with and without Attendre, we derive a forwarding delay model and numerically analyze the benefits on forwarding delay that Attendre can achieve under various controller processing delays and controller-to-switch link delays.

The rest of this paper is organized as follows. In Section II, we briefly review the concepts and terminologies in OpenFlow. In Section III, we illustrate the race conditions in OpenFlow and their ill effects. We present the Attendre mechanisms in Section IV. Section V presents the implementation of Attendre. Sections VI, VII, and VIII show the experimental results that quantify the benefits of Attendre on verification and packet forwarding delay. We present related work in Section IX. We conclude in Section X.

TABLE I
NOTATIONS USED IN THIS PAPER

Notation	Meaning
hn	nth host
sn	nth switch
ctrl	OpenFlow controller
pkt$n$	nth data packet
pkt-in$n$	nth packet-in message
fl-md$n$	nth flow-mod message
fl-md/pkt-out$n$	nth flow-mod/packet-out message

II. OPENFLOW BASICS

SDN separates the data plane and the control plane of a network. The data plane, usually the switch ASIC chip in a hardware switch platform or the kernel module in a software virtual switch, is responsible for fast packet processing, such as forwarding or dropping packets and modifying packet headers. The control plane, on the other hand, configures the packet processing rules on the data plane and handles the packets when the data plane does not know how to process them.

OpenFlow is the *de facto* standard SDN protocol the control plane uses to configure the network data plane. More specifically, configuring the data plane is achieved by sending OpenFlow command messages between a logically centralized controller and the switches. An OpenFlow switch has multiple ***flow tables***, each of which contains multiple flow table ***entries***. Each entry includes ***match fields***, a ***priority value*** and ***actions***. The match fields are defined in the OpenFlow specification. They consist of particular fields of the packet header, like MAC address and IP address, and the ingress port of the packet. A packet starts matching at flow table 0 of a switch and collects ***actions*** in the matched entries to the ***action set*** throughout the matching process. The action set is an ordered list associated with each packet and it contains the actions that will process the packet at the end of the matching process.

An OpenFlow switch communicates with the controller over a secured TCP channel. In OpenFlow, if a packet does not match with a flow table, the packet will be buffered[3] at the switch and will be associated with a ***buffer ID***, which can be used to retrieve the packet from the buffer. Then, a ***packet-in*** message containing the ***match fields*** of the packet, buffer ID and the ***table ID*** of the flow table at which the mismatch happens will be sent from the switch to the controller. By processing the packet-in, the controller could add one or more entries to the specified flow tables of switches by sending ***flow-mod*** messages to them. The controller could also issue a ***packet-out*** message containing actions and the buffer ID in the packet-in message back to the switch. The switch will process this packet according to the actions in the packet-out message. The flow-mod could also carry a buffer ID, so that a packet-out message is combined with the flow-mod. We denote this flow-mod message as a ***flow-mod/packet-out*** message. We do not differentiate flow-mod from flow-mod/packet-out in some cases, but the context in which they appear should make the meaning clear.

[2]Note that other types of race conditions exist in OpenFlow. However, they are less common and we do not present them due to space limitation.

[3]The switch can also send the entire packet to the controller together with the packet-in message.

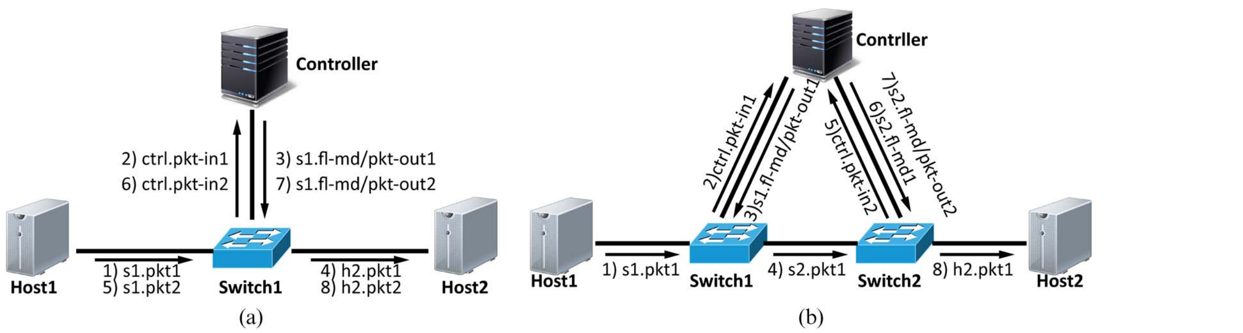


Fig. 1. Example scenarios showing race conditions in the OpenFlow protocol. (a) Network diagram for one switch and two packets. (b) Network diagram for two switches and one packet.

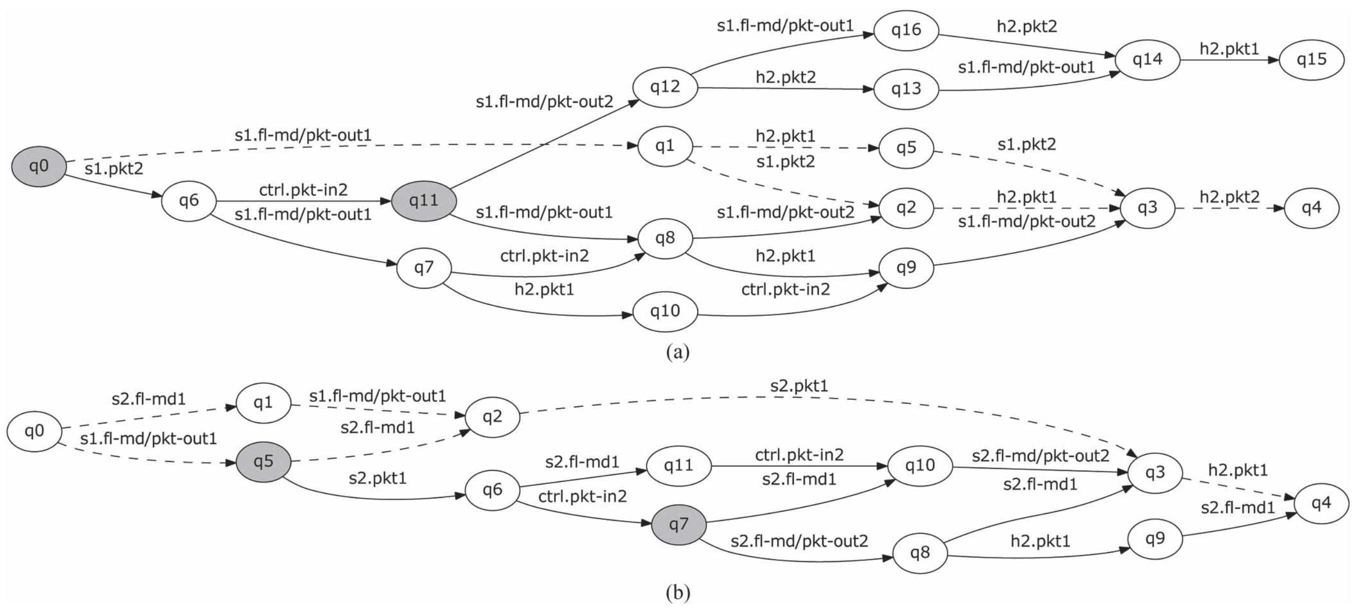


Fig. 2. Partial state diagrams for the scenarios in Fig. 1. (a) OpenFlow state diagram depicting race type 1 and 3. (b) OpenFlow state diagram depicting race type 2 and 3.

III. RACES IN OPENFLOW AND THEIR ILL EFFECTS

In this section, we reveal different types of races and their implications in detail. The following discussions assume that before the initial packet arrives at the first switch, the flow table entries that match the packet are absent in the switches. This is a common case in a network that implements fine-grained flow authentication and security management. For example, Ethane [13] and Resonance [7] only install rules for a flow when the host starts a new connection or the host sending the packet just joins the network, since each new flow or host address needs to be authenticated by the controller before the corresponding packets are allowed to traverse the network. The notations used in this discussion are shown in Table I.

The controller application we assume for this discussion is the canonical "route flow" application. This application installs forwarding entries to all the switches on the forwarding path of each packet. In detail, the application issues a flow-mod/packet-out message to the switch that sent the packet-in message (first switch) and flow-mod messages to the following switches that are on the remainder of the forwarding path.

A. Race Conditions in OpenFlow

We have identified three races prevalent in an OpenFlow network. The network topologies and the state diagrams in Figs. 1 and 2 are shown here to help explain the races. Only part of the complete state diagrams are shown for brevity.

1) Type 1: This race happens at the first switch and the race is between the processing of the non-initial data packets and the processing of the flow-mod messages containing the flow table entries addressing those packets. Fig. 1(a) is used to illustrate this type of race with one switch in the network and two packets sent from $h1$ to $h2$. $pkt1$ mismatches the flow table on $s1$ (Label 1), so $s1$ sends $pkt-in1$ to $ctrl$. Label 2 here means $ctrl$ has processed $pkt-in1$ for $pkt1$, which is represented by state q_0 in Fig. 2(a). Having processed $pkt-in1$, $ctrl$ sends $fl-md/pkt-out1$ to $s1$. Meanwhile, $h1$ also sends $pkt2$ to $h2$, racing against $fl-md/pkt-out1$.

There are two possible outcomes depending on whether $s1$ processes $fl-md/pkt-out1$ (Label 3) or $pkt2$ (Label 5) first. If Label 3 happens before Label 5, $s1$ will have a flow table entry for $pkt2$. When $pkt2$ arrives at $s1$, it will be processed according to the just installed entry, so that the state diagram takes the transitions from $q_0 \rightarrow q_1 \rightarrow \ldots \rightarrow q_4$. However, consider the other case where $pkt2$ is processed by $s1$ before $fl - md/pkt - out1$. In this case, $ctrl$ will process a redundant packet-in message from $s1$ for $pkt2$, which means Label 6 will take place according to Fig. 1(a). The state diagram in Fig. 2(a) takes the transitions from $q_0 \rightarrow q_6 \rightarrow \ldots \rightarrow q_4/q_{15}$. In addition, $pkt2$ will be processed by $s1$ only after the switch processes $fl - md/pkt - out2$ (Label 7). Thus, $pkt2$ suffers an extra delay before being forwarded.

2) Type 2: This race is experienced between data packets and flow-mod messages addressing these packets but not at the switch sending the packet-in message. Fig. 1(b) is used to illustrate this type of race with two switches and one injected packet. Fig. 2(b) is part of the state diagram for this case. Label 2 and state q_0 in Figs. 1(b) and 2(b) respectively depict the same event as described in the type 1 race. Since there are two switches in the network, $ctrl$ will also send $fl - md1$ to $s2$ to install a forwarding path for $pkt1$. After $s1$ processes $fl - md/pkt - out1$, $pkt1$ will be sent to $s2$ (Label 4). Meanwhile, if $s1$ processes $fl - md/pkt - out1$ (Label 3) before $s2$ processes $fl - md1$ (Label 6), represented by q_5 in Fig. 2(b), $fl - md1$ is still on its way to $s2$, racing against $pkt1$.

If Label 6 comes before Label 4, $pkt1$ will be addressed by the just inserted flow table entry on $s2$ by the state transitions from $q_5 \rightarrow q_2 \rightarrow q_3 \rightarrow q_4$ in Fig. 2(b). Consider the case where Label 4 happens before Label 6. $pkt1$ cannot match the flow table and a $pkt - in2$ message will be sent by $s2$ (Label 5). This $pkt - in2$ is redundant since the forwarding rule has already been sent or is going be sent to $s2$ via Label 6 and it also introduces an extra delay to the forwarding of $pkt1$. Fig. 2(b) shows the corresponding state transitions for this case from $q_5 \rightarrow q_6 \rightarrow \ldots \rightarrow q_4$.

3) Type 3: This race is experienced between different command messages issued by a multi-threaded controller for the same switch. This happens due to the controller threads competing for the same set of resources like processors or memories. In Fig. 1(b), this race happens at q_7, which represents the state where $s2$ processes $pkt1$ (Label 4) before it processes $fl - md1$ (Label 6), so that $s2$ sends $pkt - in2$ and it has been processed by $ctrl$ (Label 5). In this case, it is possible that the thread responsible for sending $fl - md1$ is delayed by the controller. Thus, it is possible that $s2$ will process $fl - md/pkt - out2$ before $fl - md1$ and vice versa. In the former case, the state diagram in Fig. 2(b) will take the state transitions $q_7 \rightarrow q_8 \rightarrow \ldots \rightarrow q_4$. But, if $s2$ processes $fl - md1$ first, the state transitions $q_7 \rightarrow q_{10} \rightarrow \ldots \rightarrow q_4$ will be taken. The same type of race happens at q_{11} in Fig. 2(a). As we can see, this race also results in many more state transitions.

B. Ill Effects Due to the Race Conditions

1) Increased Complexity in SDN Application Verification: These races result in many possible event orderings. In addition, the unnecessary command messages also add more possible events into the system. In model checking-based application verification, the verifier needs to explore every possible event ordering in the system.[4] The state diagrams shown in Fig. 2 are for the simplest scenarios, where there are no more than two switches or packets. However, when the network scales up and has more switches and more packets going through it, these races could result in a state explosion problem. The hollow markers in Fig. 8 show the numbers of state transitions explored and the time spent during the verification of the "route flow" application with various numbers of switches and packets. We use NICE [10], a model checking-based OpenFlow application verifier to collect these data. We can see that when the number of switches or packets increases, the numbers of state transitions and the verification time increase at an alarming rate. It is important to note that NICE assumes a single-threaded controller, therefore the numbers are highly conservative. If a multi-threaded controller model is used in NICE, the type 3 race will also be introduced and will cause further explosion in the state space.

2) Increased Forwarding Delay of Packets: In OpenFlow, a packet that incurs a packet-in will be processed by that switch only after the switch has processed the corresponding packet-out message. The latency from the switch sending the packet-in to the switch receiving the packet-out is composed of the round trip time between the switch and the controller and the controller processing delay. In the worst case, this happens at every hop on the path. However, the packet can be addressed by the switch as soon as the related command messages have been processed by the switch. Thus, an unnecessary packet-in that results from a race increases the forwarding delay of the packet significantly since in some cases the corresponding command messages would have been processed by the switch just after a packet-in has been sent to the controller.

3) Increased Processing Workload on the Switches and the Controller: The unnecessary packet-in, flow-mod and packet-out messages caused by the races increase the workload on both the switches and the controller. These redundant messages will further limit the scalability of an OpenFlow controller and also increase the switch processing time.

C. A Straw-Man Solution to the Type 2 Race in OpenFlow

In the OpenFlow protocol, the application could use ***barrier messages*** to avoid the type 2 race. More specifically, the controller application could hold the flow-mod/packet-out for the first switch and send a ***barrier request*** message following each flow-mod for the subsequent switches. A switch receiving a barrier request will send a ***barrier reply*** back to the controller after it finishes processing all the command messages received before the barrier request. The controller application will send the flow-mod/packet-out back to the first switch after it receives all the barrier replies from the following switches. As a consequence, when the packets reach the following switches, the flow table entries for them have already been installed. Thus, the type 2 race is eliminated.

[4]The verification approach we focus on in this paper is model checking-based verification, which will be called "verification" in short unless otherwise mentioned.

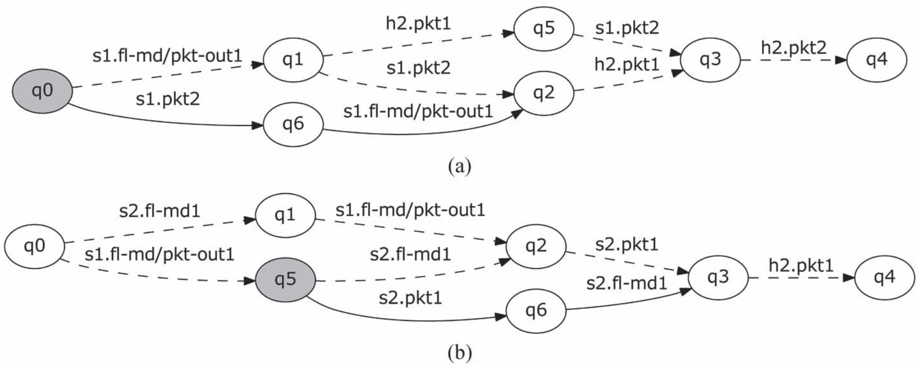


Fig. 3. Partial state diagrams for Attendre. (a) Attendre state diagram for Fig. 1(a). (b) Attendre state diagram for Fig. 1(b).

However, using barrier messages does not eliminate the type 1 and type 3 races. In addition, the waiting at the controller introduces delays to packet forwarding. We will evaluate the drawback of using barrier messages in Sections VI, VII, and VIII. In the following sections, **OpenFlow-B** will be used to denote the scenarios where the OpenFlow application uses barrier messages in the way we discussed above.

IV. Attendre

A. Overview of Attendre

As we discussed in Section III, the race conditions in OpenFlow result in many ill effects. To mitigate these ill effects, we develop Attendre, which extends the OpenFlow protocol with a series of mechanisms. The key idea behind Attendre is that it controls the outcomes of the race conditions and effectively always makes the command message win the first two types of races.

In more detail, a switch avoids sending unnecessary packet-in messages, e.g., Label 6 in Fig. 1(a) and Label 5 in Fig. 1(b), by buffering the data packets at the switch until the switch processes the flow-mod and packet-out messages for the packets. It may at first seem counter-intuitive, but the discussion in this section and the experiments in Sections VII and VIII will show that the packet buffering *decreases* the forwarding delay of a packet. By performing this packet buffering, each new flow only generates one packet-in message. It also avoids other unnecessary command messages, e.g., Label 7 in Fig. 1(a) and Label 7 in Fig. 1(b). To inform an adjacent switch of the need to buffer a flow's packets, a switch passes an Attendre Command (AC) to the next hop switch on the forwarding path of the flow. The AC tells this switch which flow it should buffer. In Attendre, the command messages sent to the same switch in response to a packet-in message form a Message Set (MS). Each MS has a unique version number and a size indicating the number of messages in the MS. These values are carried by the command messages. When a switch processes a command message, it uses these values to determine whether it has processed all the messages in the MS. After a switch has processed all the command messages in an MS, it releases the corresponding flow's buffered packets.

B. Attendre in Simple Examples

In this part, we use the two scenarios in Fig. 1 to explain how Attendre works and how Attendre reduces the cost in SDN application verification, packet forwarding delay and control plane overhead.

When a packet fails to match the flow table, if the switch is expecting to receive command messages addressing the packet, the switch should buffer the packet in its buffer. Fig. 3 gives the intuition on how Attendre works by showing the state diagrams for the network scenarios in Fig. 1. Dashed edges are the common state transitions shared between Figs. 2 and 3. Comparing the state diagrams for OpenFlow with and without Attendre, we can directly see that many state transitions are eliminated by Attendre.

1) 1 Switch & 2 Packets: With Attendre, the state diagram of this case is changed from Figs. 2(a) and 3(a). The shaded q_0 has the same meaning as that in Fig. 2(a). At q_0, where the type 1 race happens, if we take the branch where $s1$ processes $fl-md/pkt-out1$ before it processes $pkt2$, $pkt2$ will find an entry in the flow table, then $pkt2$ will be addressed by the matched entry. The dashed edges in Fig. 3(a), $q_0 \rightarrow q_1 \rightarrow \ldots \rightarrow q_4$, show the state transitions of the above process. With Attendre, when $pkt1$ is sent out via $pkt-in1$, the switch will extract the match fields of $pkt1$ and know that the controller is processing the packets having the match fields of $pkt1$. The match fields are defined in the OpenFlow specification. They consist of particular fields of the packet header, like MAC address, IP address and TCP port, and the ingress port of the packet. When the match fields of $pkt1$ arriving at $s1$ are consistent with those of $pkt2$, the latter packet will be buffered, which is represented by the state transition of $q_0 \rightarrow q_6$. Thus, $s1$ and the controller no longer need to send or process $pkt-in2$ and $fl-md/pkt-out2$, which are possible outcomes of the type 1 race. After $s1$ processes $fl-md/pkt-out1$, which means that the entry for the packets has been installed in the flow table, $pkt1$ and the buffered $pkt2$ will be sent to $h2$ by re-matching them against the flow table and taking the actions in the matched entry. This process is represented by the state transitions from $q_6 \rightarrow \ldots \rightarrow q_4$. Verification benefits from this buffering as the races and transitions related to the second packet-in will not appear in the state diagram. A switch will forward the buffered packets as soon as the corresponding commands are obtained by the switch. This reduces the delay in processing the following packets.

2) 2 Switches & 1 Packet: The state diagram in Fig. 3(b) shows the state transitions of the scenario in Fig. 1(b) in Attendre. With Attendre, after the controller processes $pkt-in1$, the controller will inform $s1$ that it will send $fl-md1$ to $s2$. This information will be sent to $s1$ together with $fl-md/pkt-out1$. The same information will then be piggybacked by $pkt1$, which matches the entry inserted to the flow table by $fl-md/pkt-out1$. On receiving $pkt1$, $s2$ knows that the controller has sent or is about to send $fl-md1$ addressing $pkt1$. $s2$ will buffer $pkt1$ until $s2$ has processed the expected $fl-md1$. After that, the buffered $pkt1$ will match the flow table again and thereby be addressed by that entry. The corresponding state transitions are $q_5 \rightarrow q_6 \rightarrow q_3$.

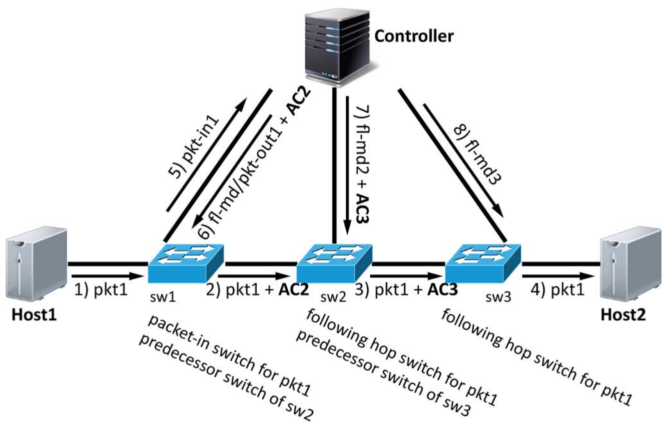

Controller
5) pkt-in1
6) fl-md/pkt-out1 + AC2
7) fl-md2 + AC3
8) fl-md3
Host1
1) pkt1
sw1
2) pkt1 + AC2
sw2
3) pkt1 + AC3
sw3
4) pkt1
Host2
packet-in switch for pkt1
predecessor switch of sw2
following hop switch for pkt1
predecessor switch of sw3
following hop switch for pkt1

Fig. 4. The propagation of Attendre command.

C. Mechanisms in Attendre—How Does Attendre Solve the Ill Effect of Race Conditions in OpenFlow?

Attendre provides a series of mechanisms to mitigate the ill effects of the race conditions in OpenFlow. These mechanisms involve minor changes to the processing procedure of the data packet, flow-mod and packet-out messages. The new processing algorithms are shown in Figs. 5, 6 and 7 respectively.

1) Addressing Type-1 Race—Buffer the Packets at the First Switch: If the first packet of a flow fails to match the flow table, the switch sends a packet-in message to the OpenFlow controller. In the rest of this section, the switch that sends the packet-in message is called the **packet-in switch** for the given flow. In Attendre, the packet-in switch avoids sending redundant packet-in messages for the following packets in the same flow, e.g., Label 6 in Fig. 1(a), by buffering these packets. Attendre provides two mechanisms to support this function:

The "*buffer*" action—In OpenFlow, a packet is processed by a list of actions of the matched flow table entry. To make the packet buffering mechanism consistent with the OpenFlow protocol, we extend the protocol with a *buffer* action.[5] This action stores the matched packet and its action-set to the switch, assigns a unique buffer ID for the packet, and inserts the buffer ID to a queue associated to the action. With the buffer ID, the switch can retrieve the packets after it processes all the flow-mod and packet-out messages for that flow.

Insert an entry with the *buffer* action—To buffer the following packets of a flow, once the first packet mismatches the flow table at the packet-in switch, the switch inserts an entry to the mismatched flow table as shown by Line 21–24 in Fig. 5. This entry contains the match fields of the first packet, is assigned the highest priority value and is given the *buffer* action. By doing so, each new flow only creates one packet-in message at the packet-in switch; when the following packets arrive at the packet-in switch, the switch buffers them.

2) Addressing Type-2 Race—Buffer the Packets at the Following Hops Switches: By saying **following hops switch**es of a flow, we mean the switches on the forwarding path of the flow not including the packet-in switch. In Attendre, if a following hop switch has not processed the flow-mod messages for the flow, that switch also buffers the packets in that flow including the first packet. To achieve this function, the following hop switch needs to know which packets it should buffer before it starts to process the packets, and buffers the packets until the switch processes the flow-mod messages for that flow. Attendre provides the following mechanisms to support this function:

The Attendre command—In Attendre, when the controller sends flow-mods to the following hops switches for a new flow, it uses the Attendre Command (AC) to tell each of the following hops switches that they may need to buffer this flow. The controller sends the AC via a special mechanism which we discuss in the following paragraph. An AC includes the match fields, a priority value, and a table ID. With these fields, when a following hop switch receives an AC, if it has not processed the flow-mod messages for this flow, the switch inserts an entry containing these fields and the *buffer* action. It is easy for the controller to generate the AC for each of the following hops switches since it is able to get the entries that will be installed on the switches by checking the flow-mod messages. For example, in the scenario in Fig. 1(b), the fields in the AC for $sw2$ are the same as the match fields, priority value and table ID of $fl-md1$.

The *encap* action—How an AC propagates in an Attendre network is shown in Fig. 4. To avoid sending packet-in messages to the controller during a flow update, a following hop switch should process the AC before the data packets in that flow, so that when the packets arrive at the switch, there is always an entry the packet can match with. This matched entry can be either a regular entry installed by the flow-mod or a *buffer* entry installed by the AC. In order to always process the AC before the packets in the flow, Attendre transmits the AC via the data plane (see Labels 2 and 3 in Fig. 4). More specifically, the AC for a switch is always encapsulated in the first data packet of the flow. When a following hop switch receives the AC from the data plane, it first extracts the AC, installs the *buffer* entry to the switch if it has not processed the flow-mod for the flow, and then processes the original data packet. (Line 3–5 in Fig. 5).

[5]The *buffer* action is an **apply-action**, which processes the packet immediately when the packet matches the entry.

```
 1: START: SWITCH receives a data packet DP
 2: if DP has encapsulated AC then
 3:     # process the Attendre Command
 4:     if SWITCH has not processed all the messages in the MS at
        AC.MS.Version then
 5:         if there is an ENTRY has the same match fields, priority and table
            id as the AC then
 6:             if AC.MS.Version > ENTRY.MS.Version then
 7:                 replace ENTRY.action with buffer action
 8:             else if AC.MS.Version = ENTRY.MS.Version then
 9:                 append buffer action to ENTRY.action
10:             end if
11:         else
12:             add a buffer entry with AC
13:         end if
14:     end if
15: end if
16: # process the original data packet
17: while initial matching or goto next flow table action do
18:     if DP matches the current flow table then
19:         execute actions in the matched entry, i.e., write actions to the action
            set of DP
20:     else
21:         # process mismatched data packet
22:         add an entry with the match fields of DP, highest priority, buffer
            action to the current flow table
23:         send packet-in for DP to the controller
24:         return
25:     end if
26: end while
27: execute the action set of DP
28: return
```

Fig. 5. Procedure for processing a data packet.

```
 1: START: SWITCH receives a flow-mod message FM
 2: if FM contains a packet, i.e., flow-mod/packet-out then
 3:     if FM has an AC then
 4:         append encap action to the action set of the packet in FM
 5:     end if
 6:     process the packet in FM
 7: else
 8:     if FM has an AC then
 9:         append the encap action to the action fields of the entry in FM
10:     end if
11: end if
12: add/modify the entry in FM to the flow table
13: if FM is the last message in the MS at FM.MS.Version then
14:     remove the buffer action
15:     let the packets buffered by the entry to re-match the flow table
16: end if
17: return
```

Fig. 6. Procedure for processing a flow-mod message.

```
 1: START: SWITCH receives a packet-out message PO
 2: if PO has an AC then
 3:     append encap action to the action set of the packet in PO
 4: end if
 5: process the packet in PO
 6: if PO is the last message in the MS at PO.MS.Version then
 7:     remove the buffer entry
 8:     let the packets buffered by the entry to re-match the flow table
 9: end if
10: return
```

Fig. 7. Procedure for processing a packet-out message.

To be consistent with the OpenFlow protocol, we add an *encap* action[6] that encapsulates the AC in the data packet. This action will be removed immediately from the entry once it is used, so that the AC is only encapsulated in the first packet.

In Attendre, the controller sends an AC targeting a following hop switch to its **predecessor switch** on the forwarding path. In this case, the AC is received by a switch via the control plane. The procedure for processing an AC received from the control plane is different from the procedure for processing an AC received from the data plane. If the predecessor switch is not the packet-in switch (e.g., $sw2$ in Fig. 4), the AC is carried by the flow-mod message for that switch (Label 7 in Fig. 4). Such a flow-mod message will add the *encap* action to the entry it modifies (Line 8–10 in Fig. 6). If the predecessor switch is a packet-in switch ($sw1$ in Fig. 4), the AC is carried by the flow-mod/packet-out or the packet-out message (Label 6 in Fig. 4). In this case, the *encap* action will apply to the packet in the packet-out directly, since this packet is the first data packet (Line 2–5 in Fig. 6 and Line 2–4 in Fig. 7, respectively).

3) Addressing Type-3 Race—Guarantee One Packet-in for Each New Flow: The type-3 race results from having duplicated packet-in messages. In Attendre, each new flow only generates one packet-in, so the type-3 race is eliminated.

4) Handling Multiple Flow-Mods to a Single Switch—Message Set: In Attendre, a switch removes the buffer action and lets the buffered packets to re-match the flow tables after the switch processes the command messages for the flow. In some cases, the controller application might send multiple command messages to a switch for one flow. In order to release the buffered packets at the right time, each switch needs to know how many command messages are sent to it. To address this issue, we introduce the notion of Message Set (MS). An MS is a collection of messages the controller sends to a switch in response to a packet-in message. An MS has an MS version for identification, and an MS size telling the switch the number of messages in that MS. Since the MS version is only used locally at each switch, the controller can simply assign the same version number to the MSes resulting from the processing of one packet-in message. The MS version and size are included in the ACs as well as flow-mod and packet-out messages. Knowing the MS version and size, a switch keeps track of the processing of messages in an MS, and removes the *buffer* entry of the flow and releases the buffered packets after the switch has processed all the messages in the MS. (Line 13–16 in Fig. 6 and Line 6–9 in Fig. 7, respectively).

5) Handling Flow Update—MS Version: In a flow update, the controller might send a flow-mod to modify the action field of an existing entry in the flow table. When a following hop switch gets an AC, if an entry in the switch has the same match fields and priority value as the AC, the switch only replaces the action with the *buffer* action when the MS version of the AC is higher than that of the entry; if the MS versions are the same, the *buffer* action will be appended to the entry if the switch has not processed all the flow-mod and packet-out messages in the MS specified by the MS version of the AC; otherwise, the switch ignores the AC. (Line 4–14 in Fig. 5).

6) Handling Multiple Flow Tables: In OpenFlow version 1.3, a switch has multiple flow tables in a pipeline. For such a switch with multiple flow tables, we highlight the following Attendre features: 1) At a packet-in switch, the switch inserts a *buffer* entry for the mismatched packet in the flow table that the packet mismatches with; 2) At a following hop switch, a *buffer* entry should be inserted into the **earliest flow table** the controller

[6]The *encap* action is a Write-Action, which writes the *encap* action into the actions set of the packet. The switch executes the *encap* action right before the *output* action.

modifies—the earliest flow table here means the first flow table the packet will match among all the flow tables the controller modifies in the MS; 3) When the buffered packets are released, they start matching the flow table where the *buffer* entry was inserted.

D. Attendre's Behavior Is a Special Case of OpenFlow

This section demonstrates that the packet processing behavior in Attendre is equivalent to a specific behavior allowed in OpenFlow. With this property, an Attendre network will not result in additional behavior beyond OpenFlow. Thus, applications that work correctly in OpenFlow are also correct in Attendre.

In Attendre, a switch knows the MS version and the MS size of each flow update. As long as the controller correctly sets these fields, the switch can remove the *buffer* action after it has processed all the command messages. Eventually, the entries in the flow table will be the same as that in OpenFlow. Thus, the processing behavior of the packets that have never been buffered is the same as in OpenFlow.

Next, we consider the packets that have been buffered by the switches. A packet is buffered only if the switch has not processed all of the command messages in the MS of the flow. The OpenFlow protocol does not provide any timing guarantee. This means that the packets could arrive at the switch after it has processed all the command messages. After the switch retrieves the buffered packets, it will let the packets to rematch the flow table. At this time, the flow table contains the actual entry for these packets. How the switch processes these buffered packets is the same as the way the switch processes the packets arriving after the switch has processed all of the command messages. Thus, the processing behavior of these buffered packets is allowed by OpenFlow.

V. Implementation

A. Implementation in Open vSwitch

We have implemented Attendre in Open vSwitch v1.9.3, the most recent LTS version [11], with about 500 lines of code. Open vSwitch (OVS) is a software OpenFlow switch, which consists of a *kernel* module and an *userspace* module. The kernel module emulates the OpenFlow data plane for fast packet processing. It receives packets, matches the packets against the kernel flow table, and processes the packets according to the actions defined in the matched entries. If a packet cannot find a matched entry in the kernel flow table, the packet will be sent to the userspace via inter-process communication. The userspace module implements the OpenFlow protocol. In particular, the userspace module connects to the OpenFlow controller, and creates, sends, receives and processes OpenFlow messages, e.g., packet-in, packet-out, flow-mod, and barrier request and reply. The userspace module also configures and manages the kernel flow table according to the OpenFlow messages it receives.

The Attendre features are added to the OVS userspace module. No change to the OVS kernel module is needed. More specifically, the packet buffering, encapsulation and decapsulation and the *buffer* entry insertion and deletion are all done in the userspace module. Implementing Attendre in the OVS userspace module allows Attendre to run on many commercial hardware switch platforms from companies like Marvell, Broadcom, Cavium [14] and Pica8 [15] that are compatible with the OVS userspace module. Implementing Attendre in the userspace module also allows it to use the switch CPU memory, which is typically implemented by cheap DRAM with large capacity, for packet buffering, instead of using the packet buffer on the switch ASIC chip which is usually based on expensive SRAM with very limited capacity.

B. Implementation in NOX

NOX is an OpenFlow controller platform [12]. We have implemented the controller features of Attendre in NOX. In Attendre, the controller issues the packet-out and the flow-mod messages carrying the AC if it wants to insert a *buffer* entry in the next hop switch. We have added APIs with which the application can create these new types of messages. The MS version and size are assigned to the command messages by the controller itself.

C. Implementation in NICE

NICE is a model checking based OpenFlow application verification tool. In NICE, the OpenFlow network components, such as data packets, command messages, hosts, OpenFlow switches and the NOX controller platform, are implemented as different models. We have integrated Attendre into NICE by modifying these network component models accordingly.

VI. Benefits to Model Checking Verification

We use NICE to verify a few representative applications including "route flow", MAC learning switch, and energy-efficient traffic engineering. We only use the *InstantStats* heuristic in the traffic engineering application verification so that the switch statistic messages can be delivered instantly. We compare the number of state transitions and the verification execution time between OpenFlow and Attendre.

The machine performing the verifications has a Quad-Core AMD Opteron 2393 SE CPU and 16 GB of RAM.

A. Route Flow

The verified network has a sender and a replier. The sender sends data packets and receives the same number of packets returned by the replier. We run the verification with different numbers of packets and different numbers of switches between the sender and the replier.

The number of state transitions and the execution time incurred during the model checking for different configurations (up to eight switches and one million state transitions) are shown in Fig. 8. Fig. 8(b) shares the same legend with Fig. 8(a). NICE uses a coarse time granularity of 0.2 s to measure the execution time. So the execution time never falls below 0.2 s.

Fig. 8 shows that in Attendre, the verification cost, i.e., the number of state transitions and the execution time, can be reduced by several orders of magnitude. For example, for the

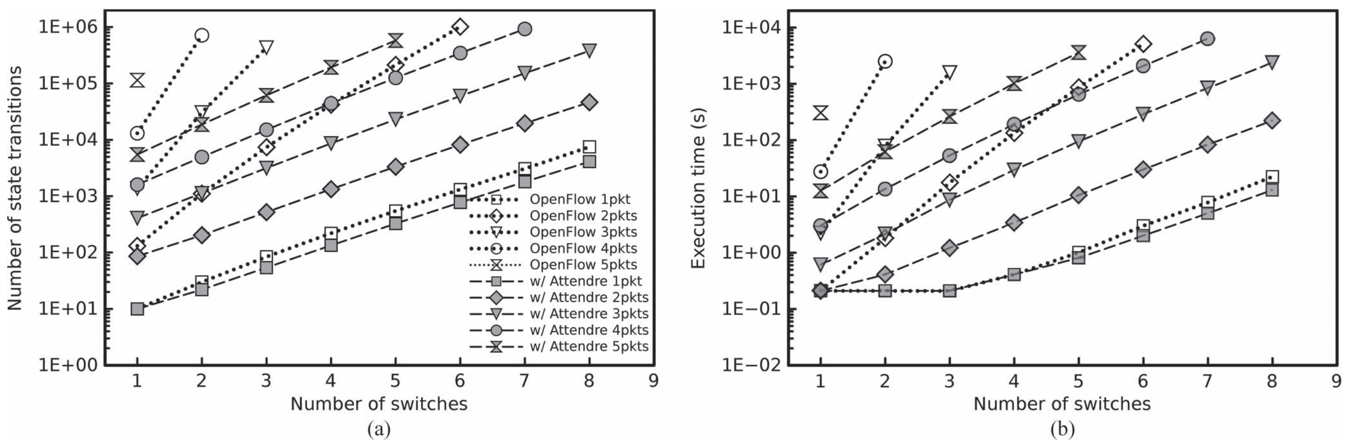


Fig. 8. NICE verification results under various scenarios for OpenFlow and Attendre. (a) Number of state transitions. (b) Execution time.

TABLE II
NUMBER OF STATE TRANSITIONS AND EXECUTION TIME IN THE MAC LEARNING APPLICATION

transition /time(second)	1 switch		2 switches	
	Attendre	OpenFlow	Attendre	OpenFlow
1 pkt	74/0.2	74/0.2	188/0.5	188/0.5
2 pkts	1.8k/2.5	2.1k/2.5	9.2k/18	11k/20
3 pkts	16.5k/27	36.1k/54	133k/355	335k/847
4 pkts	101k/209	527k/1019	-	-

TABLE III
NUMBER OF STATE TRANSITIONS AND EXECUTION TIME IN THE ENERGY-EFFICIENT TRAFFIC ENGINEERING APPLICATION

transition /time(second)	alwaysOn: 2 hops onDemand: 3 hops		alwaysOn: 3 hops onDemand: 4 hops	
	Attendre	OpenFlow	Attendre	OpenFlow
1 flow	2.1k/23	2.9k/32	8.3k/162	47.1k/1043
2 flows	27.1k/364	38.0k/497	110k/2538	642k/16473

case with two packets and six hops between the sender and the replier (diamond markers), the cost is reduced by more than two orders of magnitude (about 200 times).

We can also glean the benefits of buffering the packets at switches from Fig. 8. The benefit of buffering the non-initial packets at the first switch is shown by the experiments with only one switch. For example, in the scenario with one switch and five packets (cross markers), the cost is decreased by at least 20 times with Attendre. The benefit of buffering the initial packet at the following switches can be seen in the experiments with only one packet (square markers). For example, if there are eight switches on the path, Attendre reduces the verification cost by about 50%.

B. MAC Learning Switch

This application implements the MAC address learning protocol for an individual switch. It learns the MAC address connectivity by checking the source MAC address of the packet in each packet-in message. It also floods the packet if the destination address is unknown; otherwise, it installs a rule forwarding the packet to the learned port. The experiment uses a linear topology and varies the number of packets sent by the sender and the number of hops between the sender and replier. The replier could move to another port on the switch at a random time. Since the MAC learning application only installs rules for one hop, the network only has type 1 races.

Table II shows the verification cost (only shows the case with less than 1 M state transitions) of this application under different configurations. Attendre reduces the cost when there are more than one packet. For example, in the case with one switch and four packets, the verification cost is reduced by about five times. Table II also shows that when the number of switches increases, with the same number of packets, the reduction also increases. The verification cost for the cases with one packet remains the same between Attendre and OpenFlow, since there is no race in these cases.

C. Energy-Efficient Traffic Engineering

SDN allows the controller application to achieve online forwarding path selection for a new flow according to the traffic load on the switches and power down the unnecessary links to reduce energy consumption. We verify the REsPoNse [16] application, which pre-computes the “alwaysOn” and “onDemand” paths for each flow. When the controller receives a packet-in message, it selects the path for the flow according to the switch port statistics values, e.g., if the network is lightly loaded it chooses the “alwaysOn” path, otherwise it chooses the “onDemand” path. We use the symbolic engine in NICE to generate different port statistics. In the experiments, we vary the number of switches on the “alwaysOn” and “onDemand” paths between the sender and replier and the number of flows the sender sends. The sender only sends one packet, so only type 2 races exist in these cases.

Table III shows the verification cost. Since the controller application installs a forwarding path for the packets, the type 2 race in OpenFlow results in many redundant command messages. From Table III, we find that Attendre has a much lower verification cost than that in OpenFlow, since in all these cases, each flow only creates one packet-in message. For example, the verification cost can be reduced by 6 times when there are three switches on the “alwaysOn” path and 4 switches on the “onDemand” path. Similar to the results for the route flow application, the benefit would be even higher when the number of switches on the paths increases.

VII. Benefits to TCP Connection Setup

This experiment aims at showing that Attendre reduces the time needed to establish TCP connections. We first describe our experiment setup and then analyze the experiment results.

A. Experiment Setup

We setup a network prototype via a container-based network emulator Mininet v2.1.0 [17], with Open vSwitches and the NOX [12] OpenFlow controller platform. In this network, a client establishes a set of TCP connections with a server, i.e., the client sends TCP SYN packets with different source port numbers and receives the corresponding TCP SYN-ACK packets from the server. The experiment is done on a single desktop computer, which has an AMD Phenom II X4 965 CPU (4 cores each at 3.4 GHz) and 8 GB of RAM.

The TCP connection establishment time can be affected by the number of hops between the client and the server, and the processing delays of the switches on the forwarding path. The experiments in this section focus on studying the influence of these two factors. To compare Attendre with OpenFlow and OpenFlow using barrier messages (OpenFlow-B defined in Section III-C) in various situations, we vary the number of hops between the client and the server and the flow inter-arrival time.

We measure the round trip time (RTT) of each pair of SYN and SYN-ACK messages so as to reflect the time needed to establish a TCP connection. The time needed by the third packet, the TCP ACK, in the TCP 3-way handshake is ignored, since it consumes very little time compared to the time needed by the SYN and the SYN-ACK. This is because the ACK has the same match fields as the SYN in the same connection, and the flow table entry for them has already been installed in the OVS kernel flow table when the SYN traverses the network. Thus, the ACK will not be passed to the OVS userspace.

The goal of this experiment is to analyze the TCP connection setup time for each client-server pair in an SDN network where many clients make connections to many servers. However, from the network perspective, using only a single client-server pair to emulate the traffic has no difference from using multiple clients and servers. This is because all the packets sent by the client and the server have different port numbers. By installing exact matching rules to the switches, the controller could differentiate these packets into different flows, as if they come from multiple clients and servers. The other reason is that having only one client-server pair allows the flow arrival interval to be controlled more accurately, and also removes the unnecessary overhead of the additional hosts.

The OpenFlow controller platform we use is NOX v0.9.1 [12]. NOX provides a very small controller delay for request rates lower than 20000 requests per second [18], which is higher than the packet-in request rate in our experiment. The application running on the NOX controller is a layer 2 forwarding application that installs exact matching rules to the switches on the forwarding path in response to each packet-in. This forwarding application is based on a network topology learning application running in parallel. We have implemented this forwarding application for OpenFlow, Attendre and OpenFlow using barrier messages (OpenFlow-B) respectively.

It is worth noting that, if the controller is heavily loaded or has long processing delay, Attendre will show more benefit in packet forwarding delay. The influence of the controller processing delay will be discussed in Section VIII.

Even though the experiments are done with software OpenFlow switches, we predict that a hardware switch running OVS would show the same trends and similar results. The reason is that, as we discussed in Section V, the difference between OVS switches with and without Attendre is in the userspace. This difference is exactly the same difference between the hardware switches running OVS as the control plane.

For a switch implementing Attendre's flow table in hardware, the *buffer* and *encap* TCAM rule insertion takes about tens of memory movements [19]. Thus, the time taken by the hardware rule insertion is on the order of nanoseconds, which is comparable to software rule insertion and deletion time. We therefore expect that Attendre achieves similar performance improvement with a hardware flow table.

B. Results

In the experiment, for each case, the client establishes a thousand TCP connections with the server. Each case has a constant flow arrival interval varying from 1 ms to 32 ms and the number of hops between the client and the server varies from two to four. The CDFs of the 1000 RTTs for all the cases are shown in Fig. 9. In general, we can conclude that the more switches on the path, the higher the RTTs; and the lower the flow arrival interval, the higher the RTTs. Besides these observations, we make the following observations by comparing Attendre with OpenFlow and OpenFlow-B.

1) When the Flow Arrival Interval Is Small (Less Than 16 ms), Attendre has Much Lower RTTs Than OpenFlow: When the flow arrival interval is small, a switch in the network needs to send more packet-ins and processes more flow-mods per unit time, which increases the delay of sending or processing the messages. In **OpenFlow**, this longer delay increases the chance that a packet arrives at a following hop switch where the corresponding flow-mod has not been processed. If a packet is sent to the controller by a following hop switch, its RTT will increase significantly. Moreover, when the controller receives a packet-in from a following hop switch, it will again send a set of flow-mods back, which further increases the number of flow-mods that the switches need to process per unit time.

Attendre buffers the mismatching packet and forwards it after the corresponding flow-mod is processed. This avoids sending redundant packet-ins and reduces the number of flow-mods a switch needs to process to the minimum. Although Attendre adds a little overhead to the flow-mod processing and packet-in sending (i.e., inserting and removing the wait entries, encapsulating and de-encapsulating the packets and updating the Attendre tables), the comprehensive effect of the number of commands processed makes the workload per unit time of Attendre much less than that in OpenFlow. As a consequence, Attendre exhibits much less delay than OpenFlow. For example,

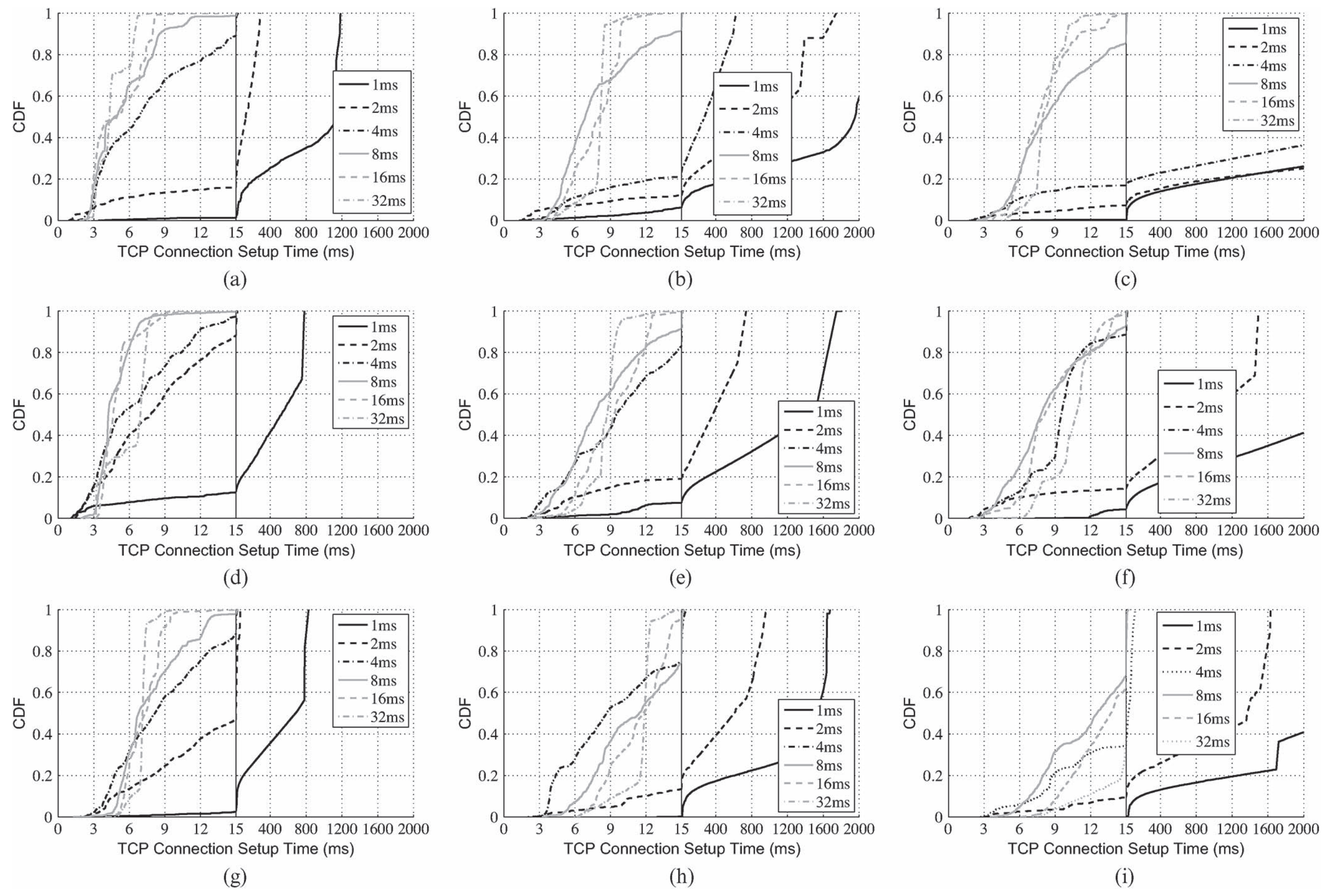


Fig. 9. CDF of TCP connection establishment time with different hop count and flow inter-arrival time. (a) OpenFlow, 2 hops. (b) OpenFlow, 3 hops. (c) OpenFlow, 4 hops. (d) OpenFlow-Attendre, 2 hops. (e) OpenFlow-Attendre, 3 hops. (f) OpenFlow-Attendre, 4 hops. (g) OpenFlow-B, 2 hops. (h) OpenFlow-B, 3 hops. (i) OpenFlow-B, 4 hops.

for the case with 3 hops and 4 ms flow arrival interval, the 80th percentile of the RTT is reduced from 500 ms (not shown in the figure) to 15 ms.

It is worth noting that in a data center network traffic study [20], the flow inter-arrival time is quite low. For example, in the top-of-rack switches in cloud data centers, more than 95% of the flow inter-arrival times are less than 10 ms; the corresponding percentage in university data centers is 60–85%. Thus, Attendre speeds up the TCP connection times for most flows in data center networks.

2) In All Cases, Attendre Has Lower RTTs Than OpenFlow-B: In OpenFlow-B, the controller sends the flow-mod/packet-out back to the first switch after it receives all the barrier replies from the following hops switches. Thus, although OpenFlow-B generates the same number of packet-ins and flow-mods as Attendre and Attendre needs a little bit more time to processing a packet-in and a flow-mod, waiting for the barrier replies adds much more delay to the RTTs. Moreover, processing the barrier requests and replies at a switch further delays the processing of other messages.

3) When the Flow Arrival Interval Is High, Attendre Has a Little Larger RTTs Than OpenFlow: For example, for the case with three switches and a flow arrival interval of 32 ms, the 80th-percentile of the RTTs in Attendre is only about 2 ms larger than that in OpenFlow. This performance degradation in Attendre is due to the fact that Attendre adds a small overhead to the processing of OpenFlow messages. On the other hand, in OpenFlow, as the flow arrival interval is high, in a specific period of time, the number of flow-mod messages a switch needs to process is low. Thus, in most cases, the following hops switches are able to finish processing the flow-mod messages before the corresponding packets arrive at these switches. As a consequence, the chance of sending packet-ins to the controller by the following hops switches is quite low in OpenFlow.

VIII. Benefits to Forwarding Delay When Controller and Network Delays Vary

This section addresses the issue of how the controller delay and the network delay between the controller and the switches affect the forwarding time of a packet. The notations used in this section are defined in Table IV. We first derive the forwarding delay formulae for a unicast packet traversing N hops for three different cases, i.e., OpenFlow, OpenFlow-B and OpenFlow-Attendre. Then, based on these formulae, we numerically analyze the forwarding delay and show that Attendre reduces the forwarding delay in most scenarios.

TABLE IV
DELAY FORMULAE NOTATIONS

Notation	Meaning
N	Number of $switches$ on the path
T^N_{case}	Forwarding delay of a specific $case$ with N switches on the path (3 cases followed)
nb	OpenFlow not using $no\ barrier$ message
b	OpenFlow using $barrier$ message
a	OpenFlow with $Attendre$ mechanism
L_k	Delay from $node_k$ pushing the packet in its output queue to $node_{k+1}$ beginning to process the packet
S	Delay from $host_0$ pushing the packet in its output queue to $controller$ beginning to process the packet
τ^k	Round-trip TCP delay between $switch_k$ and $controller$
g^k	Propagation delay between $node_k$ and $node_{k+1}$
t^k	Transmission delay from $node_k$ and $node_{k+1}$
$p^k_{s_p}$	Processing delay for $switch_k$ processing $packet$ (from matching the packet against flow table to executing the action to the packet)
$p^k_{s_c}$	Processing delay for $switch_k$ processing $command$
p^k_c	Processing delay for $controller$ processing packet-in message from $switch_k$
q^k_i	Queueing delay of the $incoming$ packet for $switch_k$
q^k_o	Queueing delay of the $outgoing$ packet for $switch_k$

A. Delay Model in OpenFlow Network

The delay in traditional network architectures is composed of four components including propagation (g^k), transmission (t^k), queueing (q^k_i/q^k_o) and processing delays. These delay components are also applicable to the delay computation of the OpenFlow network except for the processing delay containing three separate ingredients, which are $p^k_{s_p}$, $p^k_{s_c}$ and p^k_c. Besides that, the packets in OpenFlow could experience several TCP round trip time $(RTT = \tau^k)$ between $switch_k$ and $controller$, since the packet will be sent to the controller if the packet does not match the flow table.

B. Delay Formulae

In Table IV, $node$ means either a $switch$ or a $host$ in the network. Indices 0 and $N+1$ are for the sender and the receiver respectively, and indices from 1 to N are for the switches. We also define two frequently used variables in (1) and (2) to make the formulae easier to understand.

$$L_k \equiv q^k_o + t^k + g^k + q^{k+1}_i \qquad 0 \le k \le N \tag{1}$$

$$S \equiv L_0 + p^1_{s_p} + \frac{\tau^1}{2} + p^1_c \tag{2}$$

The formula for **OpenFlow** is shown in (3). Since the matched entry is absent on the switches, in all of the three cases, the packet will be sent to $controller$ by $switch_1$. Thus, if there is one switch on the path $(N = 1)$, the delay should include S, the delay of processing the packet-in, the TCP delay of sending the flow-mod/packet-out back, the delay of processing the message, and L_1. If there are more than one switch $(N \ge 2)$ on the path, the delay with N switches contains the delay with $N-1$ switches, the processing delay of the packet at $switch_N$ and L_N. Besides these, if the packet is processed by $switch_N$ before it processes the flow-mod, another RTT between the switch and the controller and the corresponding processing delays should be taken into account, which is the last term in (3). $I(b)$ is an indicator function; the value of this function is 1 when b is $true$ and 0 when b is $false$.

$$T^N_{nb} = \begin{cases} S + p^1_{s_c} + \frac{\tau^1}{2} + p^1_{s_p} + L_1 & N = 1 \\ T^{N-1}_{nb} + p^N_{s_p} + L_N + \left(\tau^N + p^N_c + p^N_{s_c} + p^N_{s_p}\right) & \\ \quad \times I\left(\left(T^{N-1}_{nb} - S\right) < \left(\frac{\tau^N}{2} + p^N_{s_c}\right)\right) & N \ge 2 \end{cases} \tag{3}$$

The formula for **OpenFlow-B** is shown in (4). The controller waiting time is the last term. The first two summations are for the delay on the forwarding path. The following four terms are the round trip time for packet-in and flow-mod and the corresponding processing delays.

$$T^N_b = \sum\nolimits_{k=0}^{N} L_k + \sum\nolimits_{k=1}^{N} p^k_{s_p} + p^1_{s_p} + \tau^1 + p^1_c + p^1_{s_c} + \max_{2 \le j \le N}\left(\tau^j + p^j_{s_c}\right) \tag{4}$$

The formula for **OpenFlow-Attendre** is shown in (5). For $N = 1$, the delay is the same as the other two cases. For $N \ge 2$, the first term in the max function is the delay from the time when $controller$ sends the flow-mod to $switch_1$ to the time $switch_k$ finishes processing the AC encapsulated packet. The second term in the max function is the delay from the time when $controller$ sends the flow-mod to $switch_k$ to the time when $switch_k$ finishes processing that flow-mod. The maximum value of these two terms plus S is the delay from the beginning to the time $switch_k$ starts processing the packet.

$$T^N_a = \begin{cases} S + \frac{\tau^1}{2} + p^1_{s_c} + p^1_{s_p} + L_1 & N = 1 \\ p^N_{s_p} + L_N + S & \\ \quad + \max\left(\left(T^{N-1}_a - S\right), \left(\frac{\tau^N}{2} + p^N_{s_c}\right)\right) & N \ge 2 \end{cases} \tag{5}$$

C. Experiment Setup

We numerically analyze the one-way forwarding time needed by a single initial packet. The word *initial* here means that the forwarding entries matching this packet are not in the flow tables of the switches, so that the controller will install the flow table entries after it receives a packet-in message. This packet could be TCP SYN packet in reality. We deliberately choose the delay values for different network delay components, such as transmission, propagation, queueing and processing, to imitate data center network scenarios. In addition, we also carefully choose the values for the TCP link delay between the switches and the controller.

The **number of switches** on the forwarding path is 6, since 6 hops can connect most servers in a medium size data center, like the 2000-node Hadoop cluster in Facebook for spam detection and ad optimization [21], with commodity 48 port switches [22] organized in a fat tree topology [23], [24]. The **transmission delays** set here are for a TCP SYN packet, since Microsoft reported that in their data center, 99.91% of traffic is TCP [22]. So the transmission delays are 3.04 μs and 0.576 μs for encapsulated and non-encapsulated SYN respectively, by assuming a 1 Gbps port speed. The per hop **propagation delay**

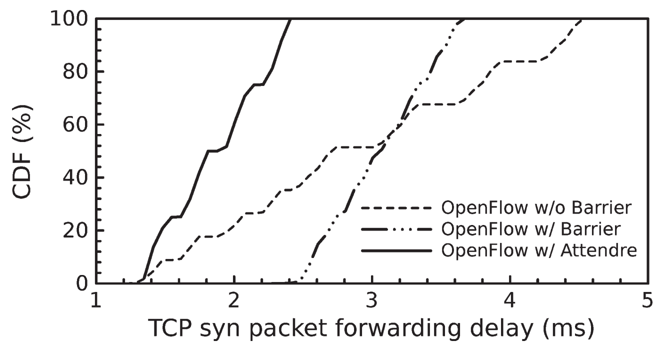


Fig. 10. Numerical analysis of delay.

in a data center is trivial comparing to other delays, since the signal propagating through a 50 m copper cable only takes 167 ns. We use this number as an approximation for the propagation delay. The **queueing delay** we use is 3.4 μs, which is half of the transmission delay for a packet with the average packet size, 850 bytes [25]. The **data packet processing delay** at the OpenFlow switch is no more than 5 μs [26]. The **command message processing delay** at the switch is about 1 ms [27].

We keep the aforementioned delays fixed throughout our computations. Then, we vary the **controller packet-in message processing delay** and the **TCP link delays** between the switches and the controller. To imitate the different controller workloads which result in various controller processing delays, we vary this value from 0.1 ms to 1 ms [18] by taking four numbers uniformly distributed in this interval. The physical position of the controller could be varied. So the round trip TCP delays between switches and the controller take on six possible values uniformly distributed between 50 μs and 250 μs, since the TCP delay in a data center is no more than 250 μs [22].

D. Numerical Analysis Results

The numerical analysis is performed on all combinations of these various delay values and applies each combination to three different OpenFlow mechanisms, OpenFlow, OpenFlow-B and OpenFlow-Attendre. Fig. 10 shows the results based on the parameters we choose and the formulae. Even though Attendre encapsulates AC to packets and buffers packets, the initial packet forwarding delay in OpenFlow-Attendre is smaller than other cases. This is because with Attendre, packets are buffered at the switches to avoid unnecessary controller processing delay, so that the packets could be processed at a switch just after the corresponding flow-mod messages are processed by the switch.

More specifically, in OpenFlow-Attendre, the initial packet forwarding delays are less than OpenFlow-B by about 1 ms. The worst case of OpenFlow-Attendre is actually better than the best case of OpenFlow-Barrier. Comparing OpenFlow-Attendre with OpenFlow, the worst case of OpenFlow-Attendre is better than 60% of the cases of the latter. This delay reduction could be very significant to certain deployment scenarios, e.g., financial institutions, where end-to-end application performance is sensitive to delays of millisecond or less. Anecdotally, the economic loss in a stock trading system could be $4 million in revenues per millisecond of extra delay [28].

IX. Related Work

A. Previous Work Related to SDN Verification

The issue of SDN verification is actively being explored in academia. Kinetic [29] proposed a way to consistently update the forwarding policies in OpenFlow networks, i.e., the packets or flows are processed by each hop with the same version of a policy. With the consistency guarantee, Kinetic simplifies the dynamic flow table configuration verification to a low cost static verification. Compared to Kinetic, Attendre has three major differences. First, the verification approach that Attendre benefits, i.e., model checking, is a more complete and rigorous verification approach than flow table configuration verification. The latter only verifies reachability properties like: is there a forwarding loop and does a packet go through a specific path, or can host A talk to host B. In contrast, model checking can verify more sophisticated invariants like: are the packets from a specific flow sent to the controller more than once, or is two packets reordered by the network. Secondly, in Kinetic, the consistency guarantee is achieved by tagging a flow table entry with a version number and tagging the version number to the packets at the first switch so that the packets could only match the entry with the tagged version number. However, when the controller installs a forwarding policy for a new flow, Kinetic changes the ingress port configuration of the first switch only after the controller knows that all of the flow-mod messages to the other switches have been installed. This way of performing network update is essentially identical to using a barrier message as described in Section III-C. We have already shown that an OpenFlow-B-like method increases the forwarding delay of packets. Thirdly, although Kinetic eliminates the ill effects of the type 2 race, it does not address type 1 and 3 races. Thus, in Kinetic, the first switch could still send redundant packet-in messages to the controller.

Another SDN verification method is run-time verification. VeriFlow [30] and NetPlumber [31] belong to this category. They check the run time behavior of applications in an operating network against packet processing invariants, and block the behavior that violates the invariants. However, these run-time tools cannot be used to verify applications before they are deployed. Thus, a potentially buggy application may be deployed live, i.e., bugs are only caught and blocked after the fact, and the network can only continue to function in a degraded mode. For these reasons, the focus of our analysis of OpenFlow has been on its suitability for model checking based application verification, an approach that is being actively developed by the NICE project [10], and is the most powerful among the existing alternatives.

Guha *et al.* proposes a machine-verified SDN controller [32] for NetCore [33], a domain specific language for SDN controller programming. This controller platform could verify if the NetCore compiler generates the correct flow table rules given a controller application written in NetCore. However, the controller does not check the correctness of the application.

B. Previous Work Aiming at Reducing the First Packet Forwarding Delay and Control Plane Workload in SDN

Reducing the first packet forwarding delay and the control plane workload have been delved into actively by the community. Different from OpenFlow, DIFANE [34], another enterprise network SDN architecture, reduces the control plane workload by pre-computing forwarding rules and distributing them to the data plane of some "authority" switches. Instead of sending a mismatching packet to the controller, the first switch will send it to an authority switch, which directly forwards the packet to its destination and installs rules back to the first switch. Thus, the first packet forwarding delay can be reduced by always keeping packets in the data plane. However, DIFANE restricts that the forwarding rule should be pre-computed and installed in the authority switch, which makes it reacts to dynamic network changes slowly. DevoFlow [35] is another modification to the OpenFlow protocol that can reduce the first packet forwarding delay and the control plane workload. In DevoFlow, switches will make local forwarding decisions for small flows, whereas only the large flow are sent to the centralized controller, which means that DevoFlow limits the visibility of the controller to only significant flows.

Both DIFANE and DevoFlow point out that a single physical controller is the bottleneck of SDN, and successfully reduce the control plane workload and the first packet forwarding delay. However, they sacrifice the ability for the controller to inspect all flows that arrive at the network. In contrast, Attendre keeps this ability unchanged. Moreover, none of them address the issue of SDN application verification efficiency.

Kandoo [36] investigates another inefficiency in the SDN control plane, i.e., processing the network events does not always use network-wide state. To relief the workload on the single centralized controller, Kandoo divides the control plane into two layers. The top layer controller runs the network application using the network-wide state, while the bottom layer of the control plane contains a group of local controllers. These local controllers run the applications only using local information which can be retrieved from a single switch. Attendre solves the problem of redundant messages in the control plane, which is orthogonal to the inefficiency that Kandoo addresses. This difference also makes Attendre compatible with Kandoo. With Attendre, both the bottom layer and the top layer controllers in Kandoo should expect less workload.

X. Conclusion

We have identified three types of race conditions present in an OpenFlow network, and presented a mechanism called Attendre that mitigates the ill effects of these race conditions. As our results indicate, the benefit of Attendre is potentially very large. More broadly, this work emphasizes the importance of considering race conditions in SDN protocol design and shows that by taking the consequence of race conditions as a first-class protocol design concern, it is possible to gain a deeper understanding of the subtle behaviors of existing protocols, and lead to improved protocol designs.

References

[1] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined WAN," in *Proc. SIGCOMM*, 2013, pp. 3–14.

[2] T. Hinrichs, J. Mitchell, N. Gude, S. Shenker, and M. Casado, "Expressing and enforcing flow-based network security policies," Univ. Chicago, Chicago, IL, USA, Tech. Rep., 2008.

[3] S. A. Mehdi, J. Khalid, and S. A. Khayam, "Revisiting traffic anomaly detection using software defined networking," in *Proc. RAID*, 2011, pp. 161–180.

[4] S. Shin *et al.*, "Fresco: Modular composable security services for software-defined networks," in *Proc. NDSS*, San Diego, CA, USA, 2013.

[5] P. Porras *et al.*, "A security enforcement kernel for openflow networks," in *Proc. HotSDN*, 2012, pp. 121–126.

[6] M. Casado *et al.*, "Rethinking enterprise network control," *IEEE/ACM Trans. Netw.*, vol. 17, no. 4, pp. 1270–1283, Aug. 2009.

[7] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: Dynamic access control for enterprise networks," in *Proc. WREN*, 2009, pp. 11–18.

[8] M. Reitblatt, M. Canini, A. Guha, and N. Foster, "FatTire: Declarative fault tolerance for software-defined networks," in *Proc. HotSDN*, 2013, pp. 109–114.

[9] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proc. SIGCOMM*, 2011, pp. 350–361.

[10] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford, "A nice way to test openflow applications," in *Proc. NSDI*, 2012, pp. 10:1–10:14.

[11] Open vswitch, 2013. [Online] Available: http://openvswitch.org/download/

[12] N. Gude *et al.*, "NOX: Towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[13] M. Casado *et al.*, "ETHANE: Taking control of the enterprise," in *Proc. SIGCOMM*, 2007, pp. 1–12.

[14] xFlow Research, Inc. [Online]. Available: http://www.xflowresearch.com/

[15] Open vSwitch (OVS) Overview. [Online]. Available: http://www.pica8.com/open-technology/open-vswitch.php

[16] N. Vasić *et al.*, "Identifying and using energy-critical paths," in *Proc. CoNEXT*, 2011, pp. 1–12.

[17] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proc. CoNEXT*, 2012, pp. 253–264.

[18] Z. Cai, A. L. Cox, and T. S. E. Ng, "Maestro: A system for scalable openflow control," Rice Univ., Houston, TX, USA, Tech. Rep. TR10-08, Dec. 2010.

[19] D. Shah and P. Gupta, "Fast updating algorithms for TCAMs," *IEEE Micro*, vol. 21, no. 1, pp. 36–47, Jan./Feb. 2001.

[20] T. Benson, A. Akella, and D. A. Maltz, "Network traffic characteristics of data centers in the wild," in *Proc. IMC*, 2010, pp. 267–280.

[21] B. Hindman *et al.*, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. NSDI*, 2011, pp. 295–308.

[22] M. Alizadeh *et al.*, "Data Center TCP (DCTCP)," in *Proc. SIGCOMM*, 2010, pp. 63–74.

[23] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proc. SIGCOMM*, 2008, pp. 63–74.

[24] A. Singla, C.-Y. Hong, L. Popa, and P. B. Godfrey, "Jellyfish: Networking data centers randomly," in *Proc. NSDI*, 2012, pp. 17:1–17:14.

[25] T. Benson, A. Anand, A. Akella, and M. Zhang, "Understanding data center traffic characteristics," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 92–99, Jan. 2010.

[26] C. Rotsos, N. Sarrar, S. Uhlig, R. Sherwood, and A. W. Moore, "OFLOPS: An open framework for openflow switch evaluation," in *Proc. PAM*, 2012, pp. 85–95.

[27] M. Jarschel *et al.*, "Modeling and performance evaluation of an openflow architecture," in *Proc. ITC*, 2011, pp. 1–7.

[28] Pragmatic Network Latency Engineering Fundamental Facts and Analysis, cPacket Networks, Mountain View, CA, USA, 2009.

[29] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker, "Abstractions for network update," in *Proc. SIGCOMM*, 2012, pp. 323–334.

[30] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. NSDI*, 2013, pp. 49–54.

[31] P. Kazemian *et al.*, "Real time network policy checking using header space analysis," in *Proc. NSDI*, 2013, pp. 99–112.

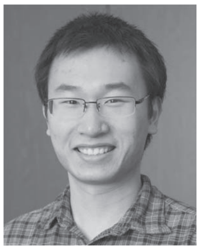
[32] A. Guha, M. Reitblatt, and N. Foster, "Machine-verified network controllers," in *Proc. PLDI*, 2013, pp. 483–494.

[33] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," in *Proc. POPL*, 2012, pp. 217–230.

[34] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, "Scalable flow-based networking with DIFANE," in *Proc. SIGCOMM*, 2010, pp. 351–362.

[35] A. R. Curtis *et al.*, "DevoFlow: Scaling flow management for high-performance networks," in *Proc. SIGCOMM*, 2011, pp. 254–265.

[36] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: A framework for efficient and scalable offloading of control applications," in *Proc. HotSDN*, 2012, pp. 19–24.



Xiaoye Steven Sun received the B.Eng. degree (with honor) in electronics engineering from Tsinghua University, in 2011 and the M.S. degree in electrical and computer engineering from Rice University, in 2014. He is currently pursuing the Ph.D. degree in electrical and computer engineering at Rice University. His research interest lies in big data systems and cloud computing.

Apoorv Agarwal received the B.Tech. degree in computer science from Vellore Institute of Technology, India, in 2009 and the M.Eng. degree in computer science from Rice University, in 2013. He is a Software Engineer at Epic.

T. S. Eugene Ng received the Ph.D. degree in computer science from Carnegie Mellon University, in 2003. He is an Associate Professor of Computer Science at Rice University. He is a recipient of a NSF CAREER Award (2005) and an Alfred P. Sloan Fellowship (2009). His research interest lies in developing new network models, network architectures, and holistic networked systems that enable a robust and manageable network infrastructure.