Parallel Simulation and Virtual-Machine-Based Emulation of Software-Defined Networks

DONG JIN, Illinois Institute of Technology
DAVID M. NICOL, University of Illinois at Urbana-Champaign

The emerging software-defined networking (SDN) technology decouples the control plane from the data plane in a computer network with open and standardized interfaces, and hence opens up the network designers' options and ability to innovate. The wide adoption of SDN in industry has motivated the development of large-scale, high-fidelity testbeds for evaluation of systems that incorporate SDN. In this article, we develop a framework to support OpenFlow-based SDN simulation and distributed emulation, by leveraging our prior work on a hybrid network testbed with a parallel network simulator and a virtual-machine-based emulation system. We show how to exploit typical SDN controller behaviors to handle performance issues caused by the centralized controller in parallel discrete-event simulation. In particular, we develop an asynchronous synchronization algorithm for passive SDN controllers and design a two-level architecture for active SDN controllers. We evaluate the system performance, showing good scalability. Finally, we present a case study, using the testbed, to evaluate network verification applications in an SDN-based data center network.

CCS Concepts: • **Networks → Network simulations**; • **Computing methodologies → Simulation tools**

Additional Key Words and Phrases: Discrete event, distributed, parallel, software-defined networking, virtual line

ACM Reference Format:
Dong Jin and David M. Nicol. 2015. Parallel simulation and virtual-machine-based emulation of software-defined networks. ACM Trans. Model. Comput. Simul. 26, 1, Article 8 (December 2015), 27 pages.
DOI: http://dx.doi.org/10.1145/2834116

1. INTRODUCTION

Manual hardware configuration is a disaster for management of modern networks, such as enterprise networks and data centers. OS and storage virtualization have already made infrastructure configurable on the fly by using software. However, networking has lagged behind, with fragmented technologies and standards. For example, to deploy a network-wide policy in a cloud platform, the network operators must (re)configure networking for thousands of physical and virtual machines, including access control lists, VLANs, and other protocol-based mechanisms. The main issue is that today's networks are operated by specialized control planes with specialized features on top of specialized hardware. Only networking vendors know how to make those things work.

This article is partly sponsored by the Maryland Procurement Office under Contract No. H98230-14-C-0141 and the Air Force Office of Scientific Research (AFOSR) under grant FA9550-15-1-0190. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Maryland Procurement Office and AFOSR.
Authors' addresses: D. Jin, Stuart Building 226E, 10 W 31st Street, Chicago, IL 60614; email: dong.jin@iit.edu; D. M. Nicol, 451 Coordinated Science Laboratory, 1308 West Main Street, Urbana, IL 61801; email: dmnicol@illinois.edu.

Traditional networking architecture is not well suited for new networking technologies, such as cloud services, mobile computing, and server virtualization.

Many switch chips already have open interfaces. Software-defined networking (SDN) aims to open up those interfaces much further with a universal interface through which either virtual or physical switches can be controlled via software. Users can create their own control planes and applications on top of the open interfaces for network management and innovation. Customized, logically centralized controllers can be designed to define the behaviors of those forwarding elements via a standardized API, such as OpenFlow [2011], which provides flexibility to define and modify the functionalities of a network after the network has been physically deployed. SDN is currently of high interest in the industry. A number of vendors, including Cisco, Dell, HP, IBM, Intel, Juniper Networks, Microsoft, NEC, and Google, are developing components and standards that enable the software-defined data center. For example, Google has deployed SDN in its largest production network: its data center–to–data center WAN [Google 2011].

The growing number of new SDN applications demand rigorous testing and evaluation before the new technologies are deployed in real large-scale networks. Since SDN is a relatively new topic, there are issues yet to be addressed, such as inconsistent views between the controller and the network, and design decisions yet to be made, such as centralized control versus distributed control, and microflow management versus aggregated-flow management. Therefore, high-fidelity testbeds for large-scale system analysis are urgently needed to facilitate the transformation of in-house research efforts of SDN to real productions. Researchers have created physical SDN testbeds, such as Ocean [University of Illinois at Urbana-Champaign 2013], to provide a realistic environment in which users can conduct (sometimes live) networking experiments. However, the network scenarios with which users can experiment have limited controllability and flexibility. Therefore, various emulation and simulation testbeds for SDN have also been developed. Emulation testbeds, like Mininet [Stanford University 2012], utilize virtualization technologies to create flexible virtual network topologies with better scalability than physical testbeds can offer. While emulation executes "unmodified software" to produce behaviors and advance experiments, simulation executes "software models." Simulation uses abstractions to accelerate changes to model states and usually requires less memory than emulation does. Hence, simulation testbeds that support SDN, like ns-3 [2011], can offer even better scalability, but the accuracy degrades because of models' simplification and abstraction.

We want to develop a testbed that combines the advantages of both emulation and simulation. In our prior work, we integrated a parallel network simulator with a virtual-machine-based emulation system [Jin et al. 2012]. When conducting network experiments, we can execute critical components in emulation and use simulation to provide a large-scale networking environment with background traffic. In this work, we develop a framework to support OpenFlow-based SDN emulation and simulation (including models of an OpenFlow switch, controller, and protocol). The new framework is based on close analysis of how SDN controllers typically behave, which led to organizational and synchronization optimizations that deal with problems that might otherwise greatly limit scalability and performance.

To improve the scalability of our testbed, we first make our emulation system distributed over multiple physical machines to support more virtual-machine-based emulated devices (around 300+ on every physical machine). Second, we develop a new global synchronization algorithm to coordinate the advancement of emulation and simulation. As a result, a larger synchronization window size is obtained, which leads to performance improvement. Third, we notice that an OpenFlow controller typically connects with a large number of OpenFlow switches. That architecture gives rise to performance drawbacks in simulating such networks using parallel discrete-event simulation. The large number of switch-controller links could potentially reduce the length of the synchronization window in barrier-based global synchronization and could also negatively impact the channel scanning type of local synchronization. Through analysis of real SDN applications, we classify the controller applications into active controllers and passive controllers based on whether the controllers proactively insert rules into the switches or the rule insertions are triggered by the switches. We design an asynchronous synchronization algorithm for the passive controllers and a two-level architecture for the active controllers for use in building scalable OpenFlow controller models. Our performance evaluation results show that the experiment execution time scales linearly as the size of the network grows.

To improve the fidelity of our testbed, we not only offer the functional fidelity by running unmodified code like many other virtual-machine-based emulations but also offer high temporal fidelity for large-scale experiments. By default, all virtual machines use the same system clock of the physical machine. As a result, when a virtual machine is idle, its clock still advances. That raises the temporal fidelity issue when applications running on a virtual machine are expected to behave as if they were being executed on a real machine. To the best of our knowledge, other existing SDN testbeds lack such temporal fidelity for scalable emulation, including the most widely used ones like Mininet [Lantz et al. 2010] and Mininet HiFi [Handigol et al. 2012]. Our SDN testbed leverages an emulation virtual time system described in our prior work [Zheng et al. 2011]. Freeing the emulation from real time enables us to run experiments slower or faster than real time. When resources are limited, we can always slow down experiments to ensure accuracy. On the other hand, experiments can be accelerated if system resources are sufficient. We also attempt to keep the high fidelity in the simulation models by using the original unmodified OpenFlow library [Stanford University 2009], which has been used to design many real SDN applications. Our testbed currently only enables the execution of real SDN software but does not support network experiments with real SDN hardware, and thus is unable to uncover SDN hardware implementation issues.

The main contributions of this article are summarized as follows:

—We develop a network testbed for OpenFlow-based SDNs, which integrates a virtual-time embedded emulation system and a parallel network simulator. The testbed supports large-scale experiments (e.g., 300+ emulated devices and tens of thousands of simulated devices on a single physical machine, and much more in distributed settings for emulated devices) and has better fidelity than Mininet.
—We design a new global synchronization algorithm for coordinating emulation and simulation to improve system scalability.
—We explore and evaluate means to improve the performance of simulation of OpenFlow controllers in parallel discrete-event simulation, including an asynchronous synchronization algorithm for controllers that are “passive” (according to a definition given in Section 5) and a two-level architecture for controllers we classify as “active.”
—We utilize our testbed to evaluate various designs of an SDN-based real-time network verifier, which detect network layer errors, such as loops and black holes, as the network states evolve.

The remainder of the article is organized as follows. Section 2 introduces background on software-defined networks, the S3F parallel simulation engine, and OpenVZ-based network emulation. Section 3 presents related work. Section 4 describes the system design. Section 5 discusses the challenges about the parallel simulation of SDN and our approaches to address the challenges. Section 6 evaluates the performance of the

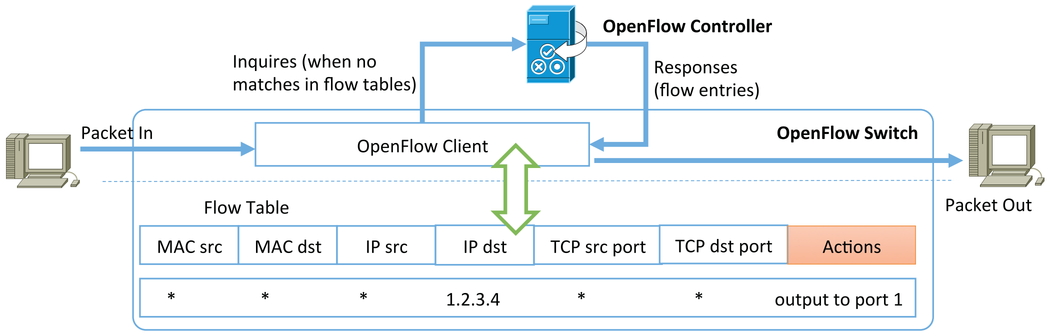

Fig. 1. Operations of a simple OpenFlow-based SDN.

testbed. Section 7 presents a case study using our testbed for an SDN-based network-wide verification system. Section 8 concludes the article with plans for future work.

2. BACKGROUND

2.1. OpenFlow-Based SDN

In a traditional network architecture, the control plane and the data plane cooperate within devices, such as switches and routers, via internal protocols. By contrast, in an SDN, the control plane is separated from the data plane, and the control logic is moved to an external controller. The controller monitors and manages all of the states in the network from a central vantage point. The controller talks to the data plane using the OpenFlow protocol [OpenFlow 2011], which defines the communication between the controller and the data planes of all the forwarding elements. The controller can set rules about the data-forwarding behaviors of each forwarding device through the OpenFlow protocol, including rules such as drop, forward, modify, or enqueue.

Each OpenFlow switch has a chain of flow tables, and each table stores a collection of flow entries. A *flow* is defined as the set of packets that match the given properties, for example, a particular pair of source and destination MAC addresses. A flow entry defines the forwarding/routing rules. It consists of a bit pattern that indicates the flow properties, a list of actions, and a set of counters. Each flow entry states, "execute this set of actions on all packets in this flow," for example, forward this packet out of port A. Figure 1 shows the main components of an OpenFlow-based SDN and the procedures by which an OpenFlow switch handles an incoming packet. When a packet arrives at a switch, the switch searches for matched flow entries in the flow tables and executes the corresponding lists of actions. If no match is found for the packet, the packet is queued, and an inquiry event is sent to the OpenFlow controller. The controller responds with a new flow entry for handling that queued packet. Subsequent packets in the same flow will be handled by the switch without contacting the controller and will be forwarded at the switch's full line rate.

Some primary benefits of applying SDN in large-scale and complex networks include the following:

—The need to configure network devices individually is eliminated.
—Policies are enforced consistently across the network infrastructures, including policies for access control, traffic engineering, quality of service, and security.
—Functionality of the network can be defined and modified after the network has been deployed.
—Addition of new features does not require change of the software on every switch, whose APIs are generally not publicly available.

2.2. S3F—Parallel Simulation Engine

The Scalable Simulation Framework (SSF) [SSF 1999] is an API developed to support modular construction of simulation models in such a way that potential parallelism can be easily identified and exploited. Following 10 years of use, we created a second-generation API named *S3F* [Nicol et al. 2011]. The S3F API contains only four main base classes, which can be used for building a complex simulation model. Their functions can be described briefly as follows:

—*Entity* is the base class that represents a simulation entity. Simulation experiments can actually be viewed as interactions among a number of entity objects. An entity object is a container of simulation state variables and instances of other simulation objects, such as OutChannels and InChannels. For example, switches and hosts are typically modeled as entities in a network simulation. An OpenFlow switch entity can contain state variables, such as the flow table and the output queue size of a network interface. An entity also includes methods that provide the logic of the simulation as they modify the simulation states.
—*inChannel* represents the endpoint of a directed communication link between entities. In S3F, the communication between entities is achieved by message passing. An entity can receive messages from another entity only through an input channel.
—*outChannel* is the starting point of a communication link between entities. An output channel of an entity can be mapped to multiple input channels that belong to this or other entities. A message sent to the output channel will be delivered by the simulation kernel to all corresponding input channels that are mapped to the output channel.
—*Message* is the base class that represents events sent between entities through the communication channels. For example, messages can represent network packets in a network simulation.

The S3F API is truly generic for systems that can be modeled as a collection of objects that communicate via message passing. This type of simulation model can be automatically mapped to multiple processors for parallel processing. A simulation is composed of interactions among a number of entity objects. Each entity is aligned to a *timeline*, which hosts an event list and is responsible for advancing all entities aligned to it. Interactions between coaligned entities need no synchronization other than this event list. Multiple timelines may run simultaneously to exploit parallelism, but they have to be carefully synchronized to guarantee global causality. The synchronization mechanism is built around explicitly expressed delays across channels whose endpoints reside on entities that are not aligned. We call these *cross-timeline channels*. The synchronization algorithm creates *synchronization windows*, within which all timelines can be safely advanced without being affected by other timelines. Moreover, when S3F is used for network experiments with both emulation and simulation, we need to design efficient and correct synchronization between the two different systems (i.e., event-driven simulation and timesliced-based emulation).

2.3. OpenVZ-Based Network Emulation

OpenVZ provides container-based virtualization for Linux [OpenVZ 2006]. It enables multiple isolated execution environments, called Virtual Environments (VEs) or containers, within a single OS kernel. It provides better performance and scalability than full or para virtualization technologies, such as QEMU [2009] or Xen [2013]. A virtual environment represents a separate physical machine, which has its own process tree starting from the init process, its own file system, users and groups, and network interfaces with IP addresses. Multiple VEs coexist within a single physical machine, and they share the physical resources and the same OS kernel, which means the Linux host operating system provides all kernel services to every VE.

Emulations executing real network applications have high functional fidelity but may not have high *temporal fidelity*, because virtual machines usually use the host machine's clock. A host serializes the execution of multiple virtual machines, and timestamps on their interactions reflect this serialization. Our version of OpenVZ provides temporal fidelity by giving each virtual machine its own *virtual clock* [Zheng et al. 2011]. The key idea is to modify the OpenVZ schedulers so as to measure the time used by virtual machines in computation (as the basis for virtual execution time) and have Linux return virtual times to virtual machines but ordinary wall clock time to other processes. The OpenVZ modifications measure the time spent in bursts of execution, stop a container on any actions that touch the network, and give one container control over the scheduling of all the other containers to ensure proper ordering of events in virtual time. Modifications to the Linux kernel are needed to trap interactions by containers with system calls related to time; for example, if a container calls *gettimeofday()*, the system should return the container's virtual time rather than the kernel's wall clock time, but calls by processes other than OpenVZ's processes ought to see the kernel's unmodified clock time. Therefore, although multiple VEs coexist on a single physical machine, they perceive virtual time as if they were running independently and concurrently.

3. RELATED WORK

OpenFlow [McKeown et al. 2008] was the first standard communications interface defined between the control and forwarding layers of an SDN architecture. Existing OpenFlow-based SDN testbeds include MiniNet [Lantz et al. 2010], MiniNet-HiFi [Handigol et al. 2012], OFTest [2011], OFlops [R.Sherwood 2011], and NS-3 [ns-3 OpenFlow Model 2011]. MiniNet is probably the most widely used SDN emulation testbed at present. It uses an OS-level virtualization technique called the *Linux container* and is able to emulate scenarios with 1,000+ hosts and Open vSwitch [2011]. However, MiniNet has not yet achieved performance fidelity, especially with limited resources, since resources are time-multiplexed by the kernel, and the overall bandwidth is limited by CPU and memory constraints. The next-generation solution, Mininet-HiFi, improves performance fidelity through CPU and link scheduling, for example, using the Linux traffic control (tc) features, but still fails to provide temporal fidelity in a virtual-machine-based environment [Heller 2013]. Ns-3 has an OpenFlow simulation model and also offers a realistic OpenFlow environment through its generic emulation capability, which has been linked to Mininet [ns-3 2013].

Researchers have incorporated various virtual machine technologies into network emulation and simulation testbeds, such as Emulab [White et al. 2002], V-eM [Apostolopoulos and Hassapis 2006], NET [Maier et al. 2007], VENICE [Liu et al. 2010], Time-Jails [Grau et al. 2008], and dONE [Bergstrom et al. 2006]. There are typically three different levels of virtualization: (1) full virtualization (e.g., VMware [1998] and QEMU [QEMU 2009]) offers complete transparency to the guest OS but with a large performance overhead, (2) para-virtualization (e.g., Xen [2013], UML [2006], and Denali [Whitaker et al. 2002]) performs modification to the guest OS to bargain for greater efficiency, and (3) OS-level virtualization (e.g., OpenVZ [2006], Virtuozzo [2012], and Linux Container [LXC 2013]) is much more lightweight and enables the guest OS to share the host kernel with independent network stacks, file systems, and process trees. Execution of real programs in virtual machines exhibits high functional fidelity, and creation of multiple virtual machines on a single physical machine provides scalability and flexibility for running networking experiments, but low temporal fidelity is a major issue for virtual-machine-based network emulation systems. Virtual machines by default use the same system clock, which implies that the time elapses even if a virtual machine is idle. Unfortunately, all the aforementioned SDN emulation testbeds are not designed to address the temporal fidelity issue.

Efforts have been made to improve temporal accuracy in virtual-machine-based network emulation and simulation. Gupta et al. [2005] modified the Xen hypervisor to translate real time into a slowed-down virtual time, running at a slower but constant rate at a sufficiently coarse timescale that makes it appear as though virtual machines are running concurrently. This is a principal technique employed in virtual machine time management named "time dilation" and has been adopted in many subsequent works in network emulation [Biswas et al. 2009; Gupta et al. 2008; Liu et al. 2010; Erazo et al. 2009; Bergstrom et al. 2006; Apostolopoulos and Hassapis 2006]. Grau et al. [2008] proposed TimeJails, a low-overhead conservative synchronization mechanism to regulate the time dilation factors. Lamps et al. [2014] extended the concept to the extremely lightweight Linux Container technology (up to 45,000 virtual machines per host). Those approaches are based on time dilation, a technique to uniformly scale the virtual machine's perception of time by a specified factor. We have taken a different approach with the focus on synchronized virtual time by modifying the hypervisor scheduling mechanism and have adopted this approach to integrate network emulation and simulation with the notion of virtual time.

Our treatment of virtual time differs from the time dilation. For example, the Xen implementations in DieCast [Gupta et al. 2008] and VAN [Biswas et al. 2009] preallocate physical resources (e.g., processor time, networks) to guest OSs. If the resources have not been fully utilized by guest OSs, the idle virtual machines (like an operating system) would simply advance the virtual clock. By contrast, we advance virtual time discretely, and only when there is an activity in the applications or network. Our approach is related to the LAPSE system [Dickens et al. 1994]. LAPSE simulates the behavior of a message-passing code running on a large number of parallel processors, by using fewer physical processors to run the application nodes and simulate the network. In LAPSE, application code is directly executed on the processors, measuring execution time by means of instrumented assembly code that counts the number of instructions executed; application calls to message-passing routines are trapped and simulated by the simulator process. The simulator process provides virtual time to the processors such that the application perceives time as if it were running on a larger number of processors. Key differences between our system and LAPSE are that we are able to measure execution time directly and provide a framework for simulating any communication network of interest, while LAPSE simulates only the switching network of the Intel Paragon. Yoginath et al. [2012] also realizes virtual time for VM-based network emulation by modifying the hypervisor scheduling. Their virtual time system is implemented on Xen and supports an individual virtual clock for each virtual core (VCPU) within a multicore VM. Our system has an emphasis on scalability by employing lightweight OS-level virtualization (OpenVZ), and only supports inter-VM scheduling to avoid additional overhead. In addition, we have integrated the virtual-time-enabled emulator to a parallel network simulator with the application of SDN.

4. SYSTEM DESIGN

We have developed a large-scale network simulation/emulation testbed. The system architecture is shown in Figure 2. The testbed consists of three major components:

—The S3F simulation engine, which is responsible for parallelizing the model execution and coordinating operations between simulation and emulation
—The S3FNet network simulator, which contains various network device, application, and protocol models, such as the OpenFlow switch, controller, and protocol models

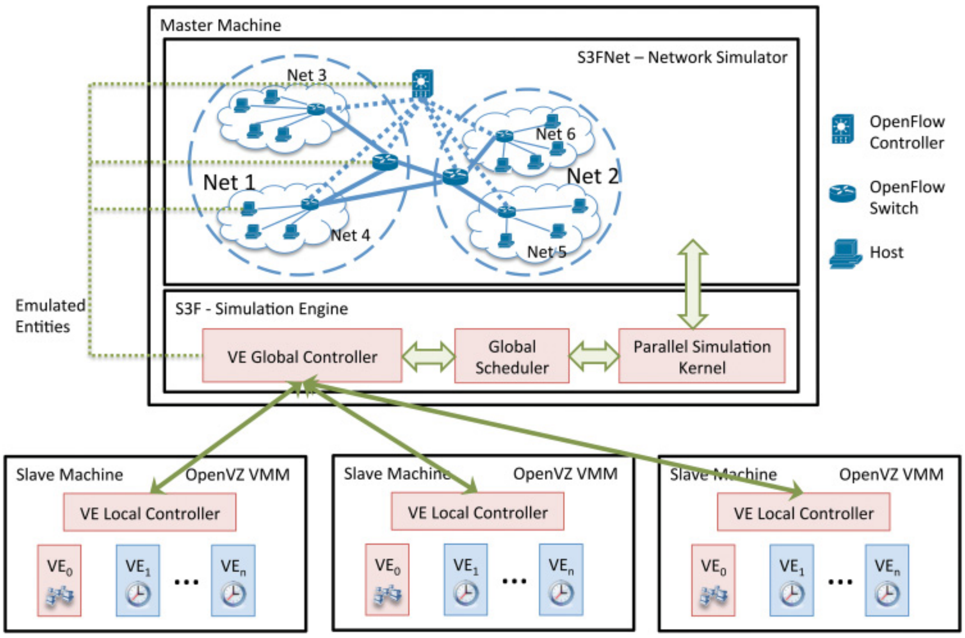

Fig. 2. S3F network testbed architecture diagram.

—The OpenVZ-based (OS-level virtualization) network emulator, whose operations are embedded in virtual time

OpenVZ enables multiple isolated execution environments within a single Linux kernel, called *Virtual Environments*. A VE runs real applications that interact with emulated I/O devices (e.g., disks) and generates and receives real network traffic passing through real operating system protocol stacks. Structurally, every VE in the OpenVZ model is represented in the S3FNet model as a host within the modeled network. Within S3FNet, traffic that is generated by a VE emerges from its proxy host inside S3FNet and, when directed to another VE, is delivered to the recipient's proxy host. The synchronization mechanism needs to know the distinction between an emulated host (VE host) and a virtual host (non-VE host). However, the type of host should make no difference to the simulated passing and receipt of network traffic. The global scheduler (discussed in Section 4) in S3F is designed for coordinating safe and efficient advancement of the two systems and to make the emulation integration nearly transparent to S3FNet.

We have been able to run millions of simulated hosts and 300+ emulated hosts on a single physical machine because of the lightweight OS-level virtualization, and one can enable more emulated hosts on a single machine simply by adding memory. To further increase the scale of the experiments the testbed can conduct, we also developed the distributed emulation capability shown in Figure 2. Currently, the parallel simulation implementation of S3F is based on shared memory, and we plan to extend S3F to distributed memory for further increasing its scalability in the future.

4.1. Emulation of Software-Defined Networks

Our testbed uses the lightweight OpenVZ containers to emulate network devices, and we are able to bring up 300+ containers in a commodity server machine. To enable experiments with larger numbers of emulated nodes, we have also extended our testbed to support distributed emulation [Zheng et al. 2013], as shown in Figure 2. A master server machine is connected to multiple slave machines via TCP/IP over gigabit Ethernet links. Each slave manages a group of local containers running on the same physical machine and independently advances program states during its emulation window. The master has knowledge of global network states. It is responsible for coordinating all slave machines through a global synchronization algorithm, for managing cross-slave events, and for performing the simulation experiments.

The global VE controller in our system (see Figure 2) is responsible for VE scheduling and message passing between VEs and simulation entities. A given experiment will create a number of guest VEs, each of which is represented by an emulation host within S3FNet. Each VE has its own virtual clock, which is synchronized with the simulation clock in S3F. The VEs' executions are controlled by the S3F simulation engine, such that the causal relationship of the whole network scenario can be preserved.

The VE controller uses special APIs to control all guest VEs. It has the following three functionalities.

—Advance emulation clock: while the VE controller communicates with OpenVZ to start and stop VE executions, it does so under the direction of the S3F global scheduler. Guest VEs are suspended until the VE controller releases them, and they can at most advance by the amount specified by S3F. When guest VEs are suspended, their virtual clocks are stopped, and their VE status (e.g., memory, file system) remains unchanged.
—Transfer packets bidirectionally: the VE controller passes packets between S3FNet and VEs. Packets sent by VEs are passed into S3FNet as simulation inputs and events, while packets are delivered to VEs whenever S3FNet determines they should.
—Provide emulation lookahead: S3F is a parallel discrete event simulator using conservative synchronization, and its performance can be significantly improved by making use of lookahead. The VE controller is responsible for providing such emulation lookahead, that is, future behavior of emulation, to the simulation engine, and the details of the lookahead are application dependent.

To emulate OpenFlow-based networks, we can run unmodified OpenFlow switch and controller programs in the OpenVZ containers, and the network environment (such as wireless or wireline media) is simulated by S3FNet. Since the executables are run on the real network stacks within the containers, the prototype behaviors are close to the behaviors in real SDNs. Once an idea works on the testbed, it can be easily deployed to production networks. Other OpenFlow emulation testbeds, like Mininet [Lantz et al. 2010], have good functional fidelity too, but lack performance fidelity, especially with heavy loads. We illustrate such differences by comparing the maximum throughput between two hosts in Mininet and the emulation system in S3F. The network contains a server–client pair connected through an OpenFlow switch, and the switch is linked to an OpenFlow controller running the learning switch application. We use iperf to generate both the TCP and UDP traffic for 10 seconds for every experiment. Two different OpenFlow switch implementations are tested, Open vSwitch [2011] and the reference implementation at Stanford [Stanford University 2009]. We conduct the experiments on an old Levono T60 laptop equipped with an Intel 2.0GHz dual-core processor, 2GB memory, and a 1-gigabit Ethernet network interface card. We repeat each experiment 10 times, and the experimental results are displayed in Table I.

It is observed that using our testbed to emulate the Stanford switch implementation can achieve significantly larger maximum throughputs than Mininet can. For TCP traffic, the maximum throughput is 1.4Gb/s, which is 2.4 times faster than the throughput in Mininet emulating OpenVSwitch and seven times faster than the throughput in Mininet emulating the Stanford switch implementation. For UDP traffic, the

Table I. S3F - Emulation and Mininet Testbeds Comparison: Single Link Maximum Throughputs

Experiment Setup		Max Throughput (Average ± Standard Deviation, Mb/s)	
Testbed	OpenFlow Switch	TCP	UDP
Mininet	OpenVSwitch (Kernel)	573 ± 5	556 ± 6
Mininet	Stanford OpenFlow Switch	194 ± 8	170 ± 3
S3F - Emulation	OpenVSwitch (User)	147 ± 23	170 ± 31
S3F - Emulation	Stanford OpenFlow Switch	1400 ± 8	1727 ± 13

maximum throughput is 1.7Gb/s, which is three times faster than the Mininet emulating OpenVSwitch and 10 times faster than the throughput in Mininet emulating the Stanford switch implementation. One limitation of our system is that the OpenVZ-based containers do not support running kernel modules. Therefore, we could only bring up the OpenVSwitch user-space module, which degrades the performance much more than the OpenVSwitch kernel module does. The experimental results show that it is not possible to emulate gigabit links even in such a simple software-defined network using Mininet on the aforementioned hardware platform. Performance fidelity will get further degraded when larger networks are emulated on a single physical machine. Imagine that one OpenFlow switch with 10 fully loaded gigabit links is emulated on a commodity physical machine with only one physical gigabit network card. There is no guarantee that a switch ready to forward a packet will be scheduled promptly by the Linux scheduler in real time. Our system does not have such limitations, since the emulation system is virtual-time embedded. The experiments can run faster or slower depending on the workload. When load is high, we can ensure performance fidelity by running the experiment slower than real time. On the other hand, when the load is low, we can reduce the execution time by quickly advancing the experiment. Note that like Mininet, our emulation system also employs OS-level virtualization technology; therefore, CPU-intensive network applications cannot be precisely emulated when the emulated devices and the host OS are required to have different CPU types. The limitation is introduced by the OS-level virtualization as a tradeoff to the system scalability.

However, experimental results show that errors introduced by the virtual-time system at the application level are bounded by the size of one emulation timeslice [Jin et al. 2012]. We may reduce the error bound by setting a smaller hardware interrupt interval [Zheng et al. 2011]. Nevertheless, the interval cannot be arbitrarily small because of efficiency concerns and hardware limits. We typically use 100μs as the smallest timeslice. If our network experiments require detailed behaviors of a gigabit switch whose processing delay is on the scale of a microsecond, emulating the switch with a 100μs error bound is simply too large. That motivated us to develop a *simulated* OpenFlow switch and an OpenFlow controller model; the simulation virtual-time unit is defined by the users and can be arbitrarily small, typically a microsecond or nanosecond.

4.2. Simulation of Software-Defined Networks

S3FNet is a network simulator built on top of the S3F kernel. It is capable of creating models of network devices (e.g., host, switch, router) with layered protocols (e.g., IP, TCP, UDP), and it can also simulate a sophisticated underlying network environment (e.g., detailed CSMA/CD for traditional Ethernet and/or CSMA/CA for wireless communication) and efficiently model the extensive communication and computation in large-scale experimental settings (e.g., background traffic simulation models [Nicol and Yan 2006; Jin and Nicol 2010]). S3FNet has a global view of the network topology. Every VE in the OpenVZ emulation system is represented in the S3FNet as a host model within the modeled network, together with other simulated nodes. Within

S3FNet, traffic that is generated by a VE emerges from its proxy host inside S3FNet and, when directed to another VE, is delivered to the recipient's proxy host, as shown in Figure 2.

We extended our network testbed to support OpenFlow-based SDN simulation experiments. Our OpenFlow simulation model consists of the OpenFlow switch and the OpenFlow controller, communicating via the OpenFlow protocol 1.0 [OpenFlow 2011]. The protocol library we use is the OpenFlow reference implementation at Stanford University [2009], which has been widely used in hardware-based and software-based SDN applications. The initial versions of the switch and the controller models have been developed with reference to the ns-3 OpenFlow Model [2011]. Our OpenFlow switch model can handle both simulated traffic and real traffic generated by the applications running in the containers. The switch model has multiple ports, and each port consists of a physical layer and a data link layer. Different physical and data link layer models allow us to simulate different types of networks, such as wireless and wireline networks. A switch layer containing a chain of flow tables is located on top of the ports. It is responsible for matching flow entries in the tables and performing the corresponding predefined actions or sending an inquiry to the controller when no match is found in the flow tables. The controller model consists of a group of applications (e.g., learning switch, link discovery) as well as a list of connected OpenFlow switches. It is responsible for generating and modifying flow entries and sending them back to the switches.

A packet is generated at the source end-host, either from the simulated application layer or from the real network application in a container. The packet is pushed down through the simulated network stacks of the host and then is popped up to the OpenFlow switch via the connected in-port. Depending on the emulation or simulation mode of the switch, the packet is searched within the simulated flow tables or the real flow tables in the container, and a set of actions are executed when matches are found. Otherwise, a new flow event is directed to the controller, meaning either the simulated controller model or the real controller program in the container. The controller generates new flow entries and installs the flow entries onto the switches via the OpenFlow protocol. Afterward, the switch knows how to process the incoming packet (and subsequent additional packets of this type) and transmit it via the correct outport. Eventually the packet is received by the application running on the destination end-host.

While running executables of OpenFlow components in the emulation mode has better functional fidelity than the simulation models do, we attempt to keep the high fidelity in the simulation models by using the original unmodified OpenFlow library, which has been used to design many real SDN applications. Also, the simulation models are not constrained by the timeslice error bound (typically set to $100\mu s$) in emulation. In addition, we can run experiments in the simulation mode with much larger network sizes. Finally, as a bonus effect of the OpenFlow design, we no longer have to preload a forwarding table at every network device, since decisions on where and how to forward the packets are made on demand by the controller. Simulating a network with millions or even more network devices at the packet level is affordable in our system.

4.3. Simulation/Emulation Synchronization

S3F synchronizes its timelines at two levels. At a coarse level, timelines are left to run during an *epoch*, which terminates either after a specified length of simulation time or when the global state meets some specified conditions. Between epochs, S3F allows a modeler to do computations that affect the global simulation state, without concern for interference by timelines. Good examples of use include periodic recalculation of path loss in a wireless simulator and periodic updating of forwarding tables within routers.

States created by these computations are otherwise taken to be constant when the simulation is running. Within an epoch, timelines synchronize with each other using barrier synchronizations, each of which establishes the length of the next synchronization window during which timelines may execute concurrently. Synchronization between emulation and simulation is managed by the global scheduler at the end of a synchronization window, when all timelines are blocked. At that point, events and control information pass between OpenVZ and S3F, using S3F's global scheduler.

We designed a global synchronization algorithm to integrate the two subsystems based on virtual time [Jin et al. 2012]. In our early design, emulation always runs ahead of simulation, and the synchronization window size is calculated as the lower bound of time when a packet may potentially reach the emulation to ensure that no emulation packets within one timeslice are missed by the simulation. However, the early design is too conservative in that it always assumes the worst-case scenarios in which a VE may send a packet immediately within a timeslice. One way to improve the system performance is to explore the emulation and simulation lookahead. In this work, we revisited the synchronization algorithm to incorporate the lookahead for speed gain. Emulation and simulation still execute their cycles alternately, but the new synchronization algorithm makes the two subsystems advance their states like a racing game. The scheduling mechanism used in the global scheduler is described in Algorithm 1, and the notations used in this section are listed next.

t_{emu}	current emulation time: OpenVZ virtual time
t_{sim}	current global simulation time, managed by S3F
ESW	emulation synchronization window: the length of the next emulation advancement
SSW	simulation synchronization window: the length of the next simulation advancement
SSW_{la}	simulation synchronization window of lookahead portion
α	a scaling factor used to model faster ($\alpha < 1$) or slower ($\alpha > 1$) processing times in OpenVZ system
TS	timeslice length in OpenVZ system, unit of VE execution time
EL_i	the event list of timeline i, a set of all events on timeline i waiting to be executed
EL_i^{emu}	a subset of events in EL_i that may affect the state of a VE, e.g., a packet delivery to a VE
EL_i^{sim}	a subset of events in EL_i that will not affect the state of a VE, $EL_i^{sim} \cup EL_i^{emu} = EL_i$
n_i	timestamp of next event in EL_i; $n_i = +\infty$ if $EL_i = \varnothing$
n_i^{emu}	timestamp of next event in EL_i^{emu}; $n_i^{emu} = +\infty$ if $EL_i^{emu} = \varnothing$
n_i^{sim}	timestamp of next event in EL_i^{sim}; $n_i^{sim} = +\infty$ if $EL_i^{sim} = \varnothing$
$w_{i,j}$	minimum "per-write delay" declared by outchannel j of timeline i "per-write delay" is a lower bound on the delay between the sending of any event from the outchannel to the mapped inchannel.
$r_{i,j,k}$	transfer time between outchannel j of timeline i and its mapped inchannel k
$s_{i,j,x}$	transfer time between outchannel j of timeline i and its mapped inchannel x, where x aligns with a timeline other than i (every inchannel is attached to a timeline during creation, see Section 2.2)
l	emulation lookahead, computed by VE controller in every ESW
emu_{flag}	a Boolean value to indicate whether the system should start to run the OpenVZ emulation; 1 means run and 0 means stop; the initial value is 1

ALGORITHM 1: Global Simulation/Emulation Synchronization Algorithm

```
while true do
    if $emu_{flag} == true$ then
        compute $ESW$;
        advance the emulation for $ESW$, i.e., $t_{emu} = t_{emu} + ESW$;
        compute the emulation lookahead;
        inject packets into the simulation;
        $emu_{flag}$ = false;
    else
        while $t_{sim} < t_{emu} + lookahead$ do
            if $t_{sim} < t_{emu}$ then
                compute $SSW$;
                advance the simulation for $SSW$, i.e., $t_{sim} = t_{sim} + SSW$;
            else if any event in any $EL_i^{sim}$ has been executed then
                $t'$ = min timestamp of such events;
                advance the simulation,
                $t_{sim} = \min(t_{emu} + \text{lookahead}, t' + \text{min channel delay})$;
                break;
            else
                compute $SSW_{la}$;
                advance the simulation for $SSW_{la}$, i.e., $t_{sim} = t_{sim} + SSW_{la}$;
        end
        $emu_{flag}$ = true;
    end
end
```

Equation (1) illustrates how ESW is calculated:

$$ESW = \max\left\{\alpha * TS,\ \min_{timeline\ i}\{P_i\} - t_{emu}\right\}, \tag{1}$$

where P_i is the lower bound of the time when an event from timeline i can potentially affect a VE-proxy entity in the simulation system, and is used by the global scheduler to decide the next ESW:

$$P_i = \min\left\{n_i^{sim} + B_i, n_i^{emu}\right\},$$

and B_i is the minimum channel delay from timeline i:

$$B_i = \min_{outchannel\ j}\left\{w_{i,j} + \min_{inchannel\ k}\{r_{i,j,k}\}\right\},$$

where $w_{i,j}$ is the minimum per-write delay declared by outchannel j of timeline i, and $r_{i,j,k}$ is the transfer time between outchannel j of timeline i and its mapped inchannel k. In our system, a packet is passed to the VE controller for delivery right after the packet is received by a VE-proxy entity in S3F. As simulation is running behind, the packet is not available to the VE controller until simulation catches up and finishes processing that event. The P_i calculation prevents a VE from running too far ahead and bypassing a potential packet delivery event.

Equation (2) illustrates how SSW is calculated:

$$SSW = \min\left\{t_{emu},\ \min_{timeline\ i}\{Q_i\}\right\} - t_{sim}, \tag{2}$$

where t_{sim} is the current global simulation time, that is, simulation end time of the previous SSW or SSW_{la}, and Q_i is the lower bound of the time that an event of timeline i can potentially affect an entity on another timeline and is used by the global scheduler to decide the next SSW:

$$Q_i = n_i + C_i,$$

and C_i is the minimum cross-timeline channel delay from timeline i:

$$C_i = \min_{outchannel\ j} \left\{ w_{i,j} + \min_{inchannel\ x} \{s_{i,j,x}\} \right\},$$

where $w_{i,j}$ is the minimum per-write delay declared by outchannel j of timeline i, and $s_{i,j,x}$ is the transfer time between outchannel j of timeline i and its mapped inchannel x, where x aligns with a timeline other than i.

Equation (3) illustrates how the simulation synchronization window for the lookahead portion, SSW_{la}, is calculated:

$$SSW_{la} = \min \left\{ t_{emu} + l,\ \min_{timeline\ i} \{R_i\} \right\} - t_{sim}, \tag{3}$$

where t_{sim} is the current global simulation time, that is, simulation end time of the previous SSW or SSW_{la}; t_{emu} is the current emulation time; l is the emulation lookahead; and R_i is the lower bound of the time that an event of timeline i can potentially affect a VE-proxy entity in the simulation system and is used by the global scheduler to decide the next SSW_{la}:

$$R_i = n_i + B_i.$$

The new algorithm has been implemented in the testbed and has been evaluated (see Section 6) and used for case study experiments. The global simulation lookahead is extracted from SDN-based application models, such as the periodic probing features in some controller applications. There are also various techniques that we could use to extract the emulation application lookahead, such as prediction based on observed input/output traffic [Zheng et al. 2013] or binary code analysis of SDN controller/switch applications. Those are interesting topics that we will explore in future work.

5. SYNCHRONIZATION CHALLENGES IN PARALLEL SIMULATION OF SOFTWARE-DEFINED NETWORKS

SDN-based network designs have multiple OpenFlow switches communicating with a single OpenFlow controller. However, many-to-one network topologies not only create communication bottlenecks at the controllers in real networks but also negatively impact the performance of conservative synchronization of parallel discrete-event simulations. The conservative synchronization approaches in parallel discrete-event simulation generally fall into two categories: synchronous approaches based on barriers [Ayani 1988; Lubachevsky 1989; Nicol 1993], and asynchronous approaches, in which a submodel's advance is a function of the advances of other submodels that might affect it [Chandy and Misra 1979]. The single-controller-to-many-switch architecture can be bad for both types of synchronization.

We can view a network model as a direct graph: nodes are entities like hosts, switches, and routers; edges are the communication links among entities; and each link is weighted by a link delay. From S3F, the parallel simulation engine's viewpoint, the graph is further aggregated: a node represents a group of entities on the same timeline, and the simulation activity of all entities on a timeline is serialized; multiple links between timelines t_i and t_j are simplified into one link whose weight is the minimum (cross-timeline) delay between t_i and t_j.

A barrier-based synchronous approach is sensitive to the minimum incoming edge weight in the entire graph. If one of the OpenFlow switch-controller links has a very small link delay (e.g., a controller and a switch could be installed on the same physical machine in reality), even if there are few activities running on the link, the overall performance will be poor because of the small synchronization window. On the other hand, an asynchronous approach focuses on timeline interactions indicated by the topology but is subject to significant overhead costs on timelines that are highly connected. The SDN-based architectures unfortunately always have a centralized controller with a large degree, which is not a desirable property for asynchronous approaches either.

To improve the simulation performance with SDN architectures, we explored the properties of an OpenFlow controller with reference to a list of basic applications in POX, which is a widely used SDN controller written in Python [POX 2011]. We have two key observations. First, controllers can be classified as either passive or active. A controller is *passive* if the applications running on it never initiate communication to switches but only passively respond to inquires from switches when no matches are found in switches' flow tables. The forwarding.l2_learning application is a good example of an application that runs on a passive controller. An *active* controller initiates communication to switches, for example, detecting whether a switch or a link is working or broken. The openflow.link_discovery application is an example of an application that runs on an active controller.

Second, a controller is not simply a large single entity shared by all the connected switches. A controller actually has states that are shared by switches at different levels. Suppose there are N switches in a network, and m switches ($1 \leq m \leq N$) share a state.

—When $m = N$, the state is network-wide; that is, the state is shared by all switches. For example, the openflow.spanning_tree application has a network-wide state, which is the global network topology.
—When $m = 1$, the state is distributed; that is, no other switch shares the state with this switch. For example, the forwarding.l2_learning application has a distributed state, which is the individual learning table for each switch.
—When $1 < m < N$, the state is shared by a subset of switches of size m. For example, the openflow.link_discovery application has such a state, which is the link status shared among all the switches connected to that link.

Based on the previous two observations, we revisited the controller design and investigated techniques to improve the performance of the simulation of OpenFlow-based SDNs with parallel discrete-event simulation. In particular, we designed an efficient asynchronous algorithm for passive controllers; we also proposed a two-level controller architecture for active controllers and analyzed performance improvement for three applications with different types of states. The proposed architecture is not only helpful with respect to simulation performance but also a useful reference for designing scalable OpenFlow controller applications.

1) Passive Controller: A passive controller indicates that applications running on the controller do not proactively talk to switches, a feature we can use in designing a controller whose functionality is known to be passive. Our new design is also motivated by another observation: when a switch receives an incoming packet, the switch can handle the packet without consulting the controller if a matched rule is found in the switch's flow table; further, for some applications, the number of times the controller must be consulted (e.g., the first time the flow is seen, or when the flow expires) is far lower than the number of packets being processed locally. All the learning switch applications in the POX controller have this property.

Therefore, our idea is that for a passive controller, switches are free to advance their model states without constraint, until the switches have to communicate with the controller. If multiple switches share a state, then the controller needs to refrain from answering the inquiring switch until all its cross-timeline-dependent switches have advanced to the time of the inquiring switch. We design an algorithm (Algorithm 2) for efficient synchronization among switches and fast simulation advancement with correct causality in the case of a passive controller, and the notations are summarized as follows:

Let A be the set of applications running in the OpenFlow controller, and R be the set of OpenFlow switches in a network model. We define $TL(r)$ to be the timeline to which switch r is aligned. For each $a \in A$ and $r \in R$, we define $f(r, a)$ to be the subset of OpenFlow switches that share at least one state with switch r for application a. For example, $f(r_1, a_1) = \{r_2, r_3\}$ means that switches r_1, r_2, and r_3 are dependent on (sharing states with) application a_1. Causality is ensured only if the controller responds to the inquiry from switch r_1 with timestamp t_1, after all the dependent cross-timeline switches r_i, that is, $r_i \in f(r_1, a)$ and $TL(r_i) \neq TL(r_1)$, have advanced their times to at least t_1. For an application with network-wide states, $f(r, a) = R - \{r\}$; for a fully distributed application, $f(r, a) = \phi$.

The algorithm is divided into two parts: one at the controller side and another at the switch side. Since the controller cannot actively affect a switch's state, it is safe for a switch to advance independently of the controller until a switch-controller event happens (e.g., a packet is received that has no match in the switch's flow tables). The delays between the controller and the switches thus do not affect the global synchronization. The causality check is performed at the controller, since it has the global dependency information of all the connected switches. Upon receiving an OpenFlow inquiry, the controller is responsible for making sure no response will be sent back to the switch (meaning that the switch will not advance its clock further) until all its dependent switches have caught up with it in simulation time. In addition, this design does not require that the controller be modeled as an S3F entity, which means that the controller does not have to align with any timeline. All the interactions can be done through function calls instead of via message passing through S3F channels. This design works only for a passive controller and can greatly reduce the communication overhead between the controller and switches. As a result, a passive controller is not a bottleneck in conservatively synchronized parallel simulation, as low latency to switches and high fan-out might otherwise cause it to be.

The algorithm works for all passive controllers, whether the state is distributed (e.g., forwarding.l2_learning) or network-wide (e.g., forwarding.l2_multi). The performance of passive controllers benefits from the use of distributed states as well as the smaller number of cross-timeline-dependent switches. In S3F, the global synchronization algorithm (see Section 4) runs beyond the synchronization algorithm for passive controllers to ensure that all switches can always advance their simulation states without encountering any deadlocks.

Our synchronization algorithm is similar to those existing asynchronous approaches [Xiao et al. 1999; Simmonds et al. 2002; Nicol and Liu 2002] in the sense that all of them focus only on timeline interactions that the topology indicates can occur, and the advancement of each timeline is dependent on the predicted future behavior of other timelines that may affect it. However, our approach is more specific to the SDN-based network architecture with a centralized controller. The controller is responsible and capable to make the scheduling decision when necessary rather than letting each timeline make its own decision through an extensive (possibly expensive) channel scanning mechanism.

2) Active Controller: Active OpenFlow controllers proactively send events to the connected OpenFlow switches, and those events can potentially affect the states of the switches. Therefore, the switches do not have the freedom to advance the model states like those switches that connect to passive controllers, but are subject to the minimum link latency between the controller and the switches. However, the question we have is:

ALGORITHM 2: Synchronization Algorithm with Passive Controller

```
Controller Side
/* Upon receiving an OpenFlow inquiry from switch $r_i$ with timestamp $t_i$ */
for each application $a_j$ related to the inquiry do
  for each switch $r_k \in f(r_i, a_j)$ AND $TL(r_k) \neq TL(r_i)$ do
    get the current simulation time, $t_k$, of $r_k$
    if $t_k < t_i$ then
      schedule a timing report event at time $t_i$ on the timeline of $r_k$
      increase $dcts[i]$ by 1
      /* $dcts[i]$ is the counter of unresolved dependent cross-timeline switches for switch $r_i$ */
    end if
  end for
end for

pthread_mutex_lock()
while $dcts[i] > 0$ do
  pthread_cond_wait()
end while
pthread_mutex_unlock()

process the inquiry (i.e., generate rules to handle packets)
send an OpenFlow response to switch $r_i$

Switch Side
/* Upon receiving a packet at an ingress port */
check flow tables
if  found matched rule(s) then
  process the packet accordingly
else
  send an OpenFlow inquiry to the controller
end if

/* On reception of an OpenFLOW response */
store the rule(s) in the local flow table(s)
process the packet accordingly

/* Scheduled timing report event for switch $r_i$ fires */
pthread_mutex_lock()
decrease $dcts[i]$ by 1
if $dcts[i] = 0$ then
  pthread_cond_signal()
end if
pthread_mutex_unlock()
```

are the assumptions about connectivity in SDNs overly pessimistic? For example, can any timeline generate an event at any instant that might affect every other timeline?

We make the following observations about the active controller applications. First, not all controllers have only network-wide states. Some have fully distributed states, for example, a switch's on/off state (openflow.keep_alive application); some have states that are shared among a number of switches—for example, a link's on/off state is shared among switches connected to the same link (openflow.link_discovery application). Second, not all events will result in global state changes, and quite often a large number of events are handled locally and only influence switches' local states. For instance, only when a communication link fails, a link is recovered from failure, or some new

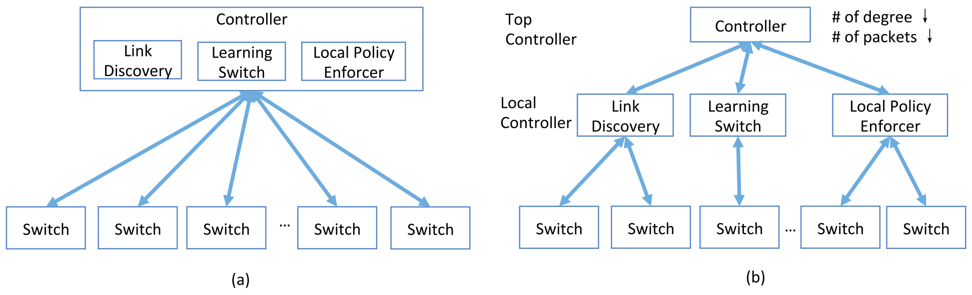

Fig. 3. (a) Typical SDN controller architecture. (b) Two-level SDN controller architecture.

switches join or leave the network is a link status change event generated for the openflow.link_discovery application, so that it updates its global view; during the remaining (most of the) time, the network-wide state, that is, the global topology, remains unchanged. Therefore, for such applications, instead of having a centralized controller frequently send messages to all the connected switches and wait for responses, we can first determine which events affect network-wide states and which do not, and then offload those events that only cause local state changes toward the switch side. Thus, we relieve the pressure at the controller.

Based on our observations of active controller applications, we modify the existing architecture (Figure 3(a)) to a two-level active controller architecture (Figure 3(b)). Local states are separated from network-wide states in a controller, and the controller is divided into a top controller and a number of local controllers. The top controller communicates only with the local controllers, not with the switches, to handle events that potentially affect the network-wide states. The local controllers run applications that can function using the local states in switches, for example, the local policy enforcer, or link discovery. There are no links among local controllers. With the two-level controller design, we aim to improve the overall scalability, especially at the top controller, as follows:

—The top controller has a smaller degree, which is good for local synchronization approaches.
—Fewer messages are expected to occur among local controllers and the top controller for many applications, since the heavy communication is kept between the switches and local controllers. That is good for local synchronization approaches as well.
—If we align the switches that share the same local controller to the same timeline, the local controllers actually do not have to be modeled as an entity. Message passing through channels is not needed, as function calls are sufficient.

Conversion of a controller into a two-level architecture requires that modelers carefully analyze the states in the controller applications. That process is not only useful in creating a scalable simulation model but also helpful in designing a high-performance real SDN controller, because it offloads local-events processing to local resources. We have studied three active applications (openflow.keep_alive, openflow.spanning_tree, and openflow.link_discovery) in the POX controller [Jin and Nicol 2013]. Through careful application state analysis, we can often convert applications with local states into two-level controller architectures for simulation performance gain. The analysis not only helps modelers to create scalable SDN network models but also helps in the design of scalable real SDN controllers. Many SDN applications can be far more complicated than the basic applications in POX, possibly a combination of passive and active applications, with both distributed and network-wide states, and the states may change dynamically with time and network conditions. Even so, it is still useful to divide a complex controller into small building-block applications, classify them according to the state type as well as passiveness/activeness, and then apply the corresponding performance optimization techniques.

6. PERFORMANCE EVALUATION

We conducted experiments to evaluate the global synchronization algorithm implemented in the testbed as well as how well the performance scales as the size of the network models grows. The testing platform for conducting all the experiments in this section was built on a Dell PowerEdge R720 server with two 8-core processors (2.00GHz per core) and 64GB RAM and installed with 64-bit Linux OS. With the hyperthreading functionality enabled, our network simulator can concurrently explore up to 32 logical processors.

6.1. Global Synchronization

Synchronization windows indicate how long the emulation or the simulation can proceed without affecting entities on other timelines. The lengths of the windows are computed at synchronization barriers based on detailed network-level and application-level information, such as minimum link delay and minimum packet transfer time along the communication paths, or network idle time contributed by the simulated devices that do not actively initiate events (e.g., server, router, switch), or the lookahead offered by the OpenVZ emulation. In this set of experiments, we wanted to investigate the performance impact of the size of the synchronization window on our simulation/emulation testbed. We set up a network with 64 emulated hosts, among which every two hosts paired up a server–client connection, and 32 links in total. Each server sent a constant-bit-rate UDP traffic flow with a constant 1,500-byte packet size to its client. The emulation timeslice is set to 1ms, and the networking environment created in S3FNet had 1Gb/s bandwidth and a 1ms link delay. We varied the sending rate with 100Kb/s, 1Mb/s, and 10Mb/s in the aforementioned scenarios and recorded both the emulation time and the simulation time every 10,000 packets. We then precalculated a constant lookahead based on the observed average interpacket gap, essentially creating larger synchronization windows, and reran the experiments for comparison. We observed nearly identical sending/receiving traffic patterns for each scenario with and without lookahead, which indicates little loss of experimental accuracy in the presence of the lookahead. Each experiment was repeated 10 times, and the average execution times for both emulation and simulation are shown in Figures 4 and 5.

With emulation lookahead, the execution time, for both emulation and simulation, was significantly reduced for the 100Kb/s and 1Mb/s sending rates. That can be explained as follows. For the 1Mb/s sending rate, given a 1,500-byte packet size, the average interpacket time is around 12ms, which equals 12 timeslices. Accurate emulation lookahead should predict that amount (i.e., promise that a VE will not generate any events within the next such amount of time) and offer that value to S3F for computing the next emulation synchronization window (ESW). The new ESW increases to approximately 12 timeslices in length and thus minimizes the emulation overhead. Since the emulation is now running far ahead of time as compared with the case without emulation lookahead, and no events are injected into the simulation's event lists, the simulation also takes advantage of the empty event lists to compute a large simulation synchronization window. Therefore, the execution times on both simulation and emulation are significantly reduced. The same is true for the 100Kb/s case. However, little improvement is observed for the 10Mb/s case, because the ESW generated by emulation lookaheads (1.2ms) is close to one timeslice. Another observation is that the accurate lookahead results in the execution times for transmitting

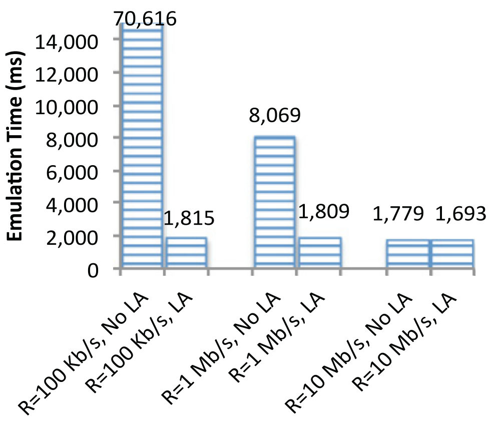

Fig. 4. Average emulation execution time (R = sending rate, LA = lookahead).

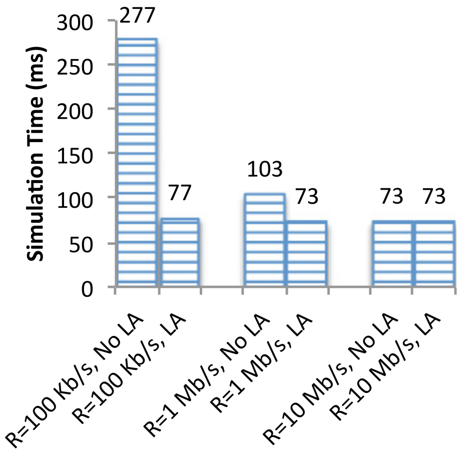

Fig. 5. Average simulation execution time (R = sending rate, LA = lookahead).

10,000 packets being very similar for all the sending rates. That is actually one benefit of our virtual-time-embedded emulation system: we can accelerate low-traffic-load emulation experiments by utilizing available system processing resources. Providing good simulation lookahead is always challenging for conservative network simulation, and our virtual-time-based simulation/emulation testbed gives another opportunity for providing good emulation lookahead. The aforementioned experimental results clearly indicate the huge performance gain that a good lookahead mechanism can bring to our system, and thus strongly motivate our ongoing work investigating other types of emulation lookahead, such as source code/binary analysis.

6.2. Performance Evaluation of Passive Controller Algorithm

In this set of simulation experiments, we explored the scalability of our testbed as well as the performance improvement with the asynchronous synchronization algorithm for the passive controllers. We created network models with 32 timelines. The backbone of the network model consisted of a number of OpenFlow switches, and each switch was connected to 10 hosts. We increased the number of OpenFlow switches connected to the controller from one up to 5,000, and the size of the network was thus increased proportionally. Half of the hosts ran client applications, and the other half ran server applications. The minimum communication link delay among the controller and switches was set to be 1ms. During the experiments, each client randomly chose a server from across the entire network and started to download files of 3KB via TCP links. Once a file transfer was complete, the client picked another server and started the downloading process again. All the OpenFlow switches connected to an OpenFlow controller, and the controller ran the openflow.learning_multi application, which is essentially a learning switch, but it learns where a MAC address is by looking up the topology of the entire network. In the first set of experiments, we modeled the controller as a normal S3F entity, and it connected to switches through channels. In the second set of experiments, we used the passive controller design described in Section 5. We repeated each experiment 10 times.

Figure 6 and Figure 7 show the average execution time and the event processing rate with standard deviation, respectively. In the entity-based controller case, the traditional barrier-type global synchronization was used. In the non-entity-based controller case, synchronization was actually two-level: the global synchronization was

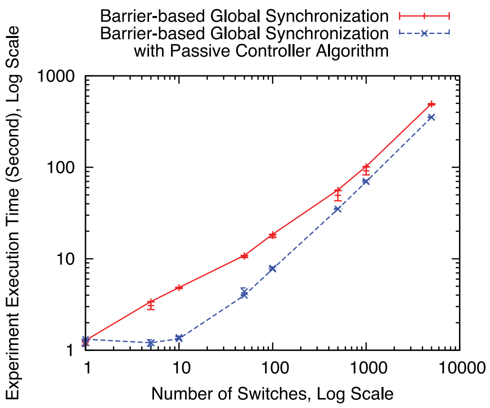

Fig. 6. Scalability: experiment execution time.

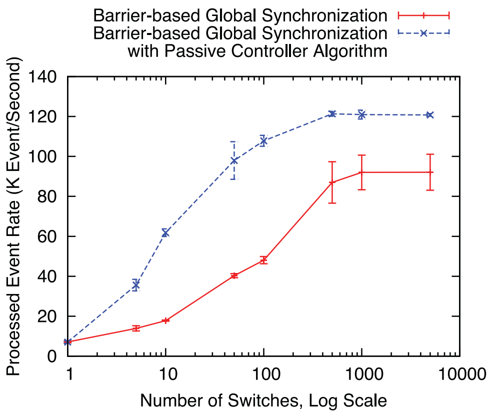

Fig. 7. Scalability: event processing rate.

used to simulate activities among switches and hosts, and within a synchronization window, the asynchronous synchronization algorithm was used to simulate interactions between the switches and the passive controller. We can see that in both cases, the average execution time scaled almost linearly, which means the performance did not degrade as the model size increased. We also notice that the non-entity-based controller case with the passive controller algorithm (in blue) always performed better than the entity-based controller case (in red) for both execution time and the event-processing rate. That performance gain is a consequence of replacing the message-passing mechanism with function calls for the controller-switch interactions. Further improvement has been observed with smaller synchronization window sizes due to smaller controller-switch delays [Jin and Nicol 2013]. The average event-processing rate increased significantly at the beginning, since the model was not yet fully parallelized over the 32 timelines because of the small network size. Once the switch size reached 1,000, the average event-processing rate stabilized at 90,000 events per second for the entity-based controller case, and 120,000 events per second for the non-entity-based controller case, which indicates a 33% improvement in terms of the average event-processing rate with the passive synchronization algorithm.

7. CASE STUDY: UNCERTAINTY-AWARE NETWORK LAYER VERIFICATION

One of the many benefits of applying SDN is that the logically centralized SDN controller has a global view of the entire network, which enables us to conduct network-wide verification, such as loop-freeness, end-to-end latency bound, and/or no-violation-of-access-control policies. Figure 8 depicts the design of our SDN-based network verifier. Sitting between the SDN controller and the network, the system intercepts and verifies every control update before the update hits the network. If any violation of security policies and network invariants is detected, alerts will be raised to tell the operations to take response actions. We initially leveraged an existing system, VeriFlow [Khurshid et al. 2013], to speed up the design process of the network-wide verifier for SDN-based networks. However, observations of a network from an SDN controller's viewpoint are always delayed at any time instance, because of the inevitable latency (such as networking delay and rule installation) between the controller and network devices. Neglecting such controller-switch delays could result in severe system performance drops. Furthermore, the controller-switch delays vary across devices and over time. After issuing updates to the network, the controller has limited knowledge of when and in what order the updates are applied. We define the inconsistency between

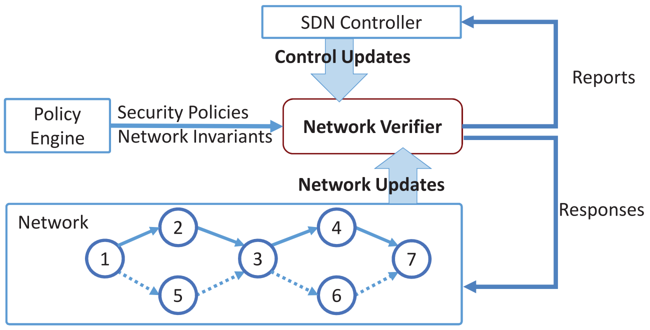

Fig. 8. Design of an SDN-based network verification system.

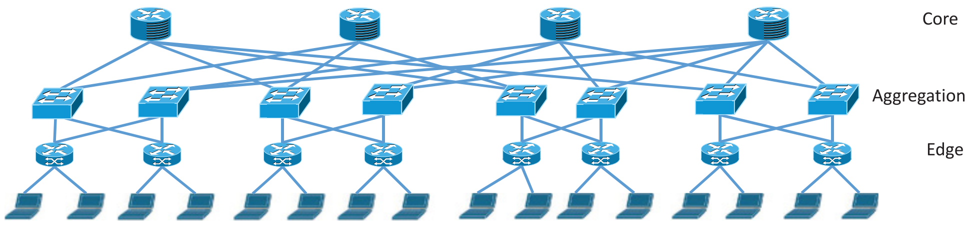

Fig. 9. A typical multirooted tree topology in an SDN-based data center network.

the view of the controller and the network state that data packets encounter as the *network temporal uncertainties*, which could lead to transient network errors and security policy violations. This motivates us to design a new uncertainty-aware network verification system. With the help of our SDN emulation/simulation testbed, we can evaluate the impact of the controller-switch latency, and our new uncertainty-aware design of the network verifier, with controllable and realistic settings.

We conducted simulation/emulation experiments to evaluate the impact of the controller-switch delay. We modeled a typical data center network interconnected by three layers of switches running the Equal Cost Multipath forwarding (ECMP) protocol as shown in Figure 9. ECMP is widely used in data center networks to take advantage of the path multiplicity by using flow hashing to statically select a path among all of the equal-cost paths. In our testbed, all the OpenFlow switches and the POX controllers running the ECMP routing application were emulated in the OpenVZ containers, and hosts and the networking environment were simulated. In the experiments, each host sent TCP traffic to any other host in the network with uniform probability, and there were 145 flows in total. We modeled the delays between the SDN controller and the switches as a uniform distribution, $U(0, n)$, and we vary n from zero up to 100ms. The experiment was repeated on a single physical machine 10 times, and the setup was relatively flexible and controllable in comparison to physical SDN testbeds.

The CDF of the flow connection establishment time is plotted in Figure 10. We notice that as n increases, that is, larger delay between the controller and the switches, it takes a longer time for flows to establish connections. When n is greater than 5ms, a dramatic increase in flow connection time is observed. Moreover, some flow connections are dropped due to TCP connection timeout, when n is greater than 2ms. We plot the average number of dropped flows (with little variance) in Figure 11. We can see that more flows are dropped as n grows (e.g., around 17% flow drop rate when $n = 9$ms). The

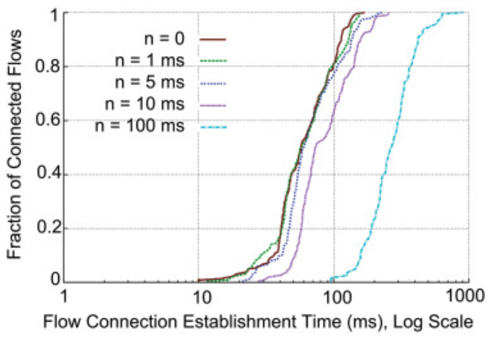

Fig. 10. Flow connection establishment time, the controller-switch latency $\sim U(0, n)$.

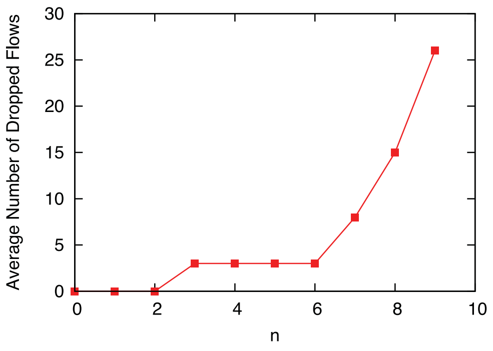

Fig. 11. Flow drop, the controller-switch latency is uniformed distributed $\sim U(0, n)$.

experimental results show the severe consequences of neglecting the controller-switch latency, which our new network verifier must address.

Next, we investigate the network temporal uncertainty issues for network verification. Short-term network faults should not be neglected for building secure and high-performance networks, such as mobile networks and data centers. For example, suppose a data center administrator issues a permit-access rule to a firewall, but later it decides to withdraw this rule. Because of the network temporal uncertainty, what could happen at the firewall is that the rule removal could be executed before the rule insertion. As a result, malicious or untrustworthy packets may enter a secure zone because of a temporary access control violation. Failing to consider network temporal uncertainty will result in many hidden errors during network state changes, such as those that occur during DDoS attacks, misconfiguration corrections, or system upgrades.

To address the challenges caused by the inevitable network temporal uncertainty, we need a new uncertainty-aware network verification design. Existing tools [Mai et al. 2011; Al-Shaer et al. 2009; Kazemian et al. 2012, 2013; Khurshid et al. 2013] verify network state by checking a snapshot of the entire network. But none of them take into account the uncertainty during the transition of the snapshots. To model the network temporal uncertainty, we symbolically modeled the forwarding behaviors of packets that arrive before and after an update occurs in the network, until the status of the update is certain (e.g., through acknowledgment or timeout). Our approach was to model the forwarding behavior as an "uncertain graph," where a subset of links are marked as uncertain links (depicted as dashed lines in Figure 8). We store all the possible forwarding rules, including rules that are supposed to be deleted or replaced by previously issued operations with pending confirmations, and mark those forwarding rules as uncertain rules. When rules are collected to form a forwarding graph at each device, the network verifier retrieves all the rules from the highest priority to lower priorities until a certain rule is found. With that approach, some networking devices may have more than one outgoing link, among which up to one is certain, and the uncertain graph is the representation of all the possible combinations of forwarding decisions at all the devices. That way, we can model the network uncertainty using a single graph and answer queries by traversing the graph once.

With the new network verifier design, the next natural question is, what is the performance improvement of the new design over the old design? For example, by how much can we improve error coverage by taking uncertainty into account? Our SDN emulation/simulation testbed has been utilized to answer those questions. We first

Table II. Errors in Error Detection by Network Verifiers, Two-Level Fat-Tree with Learning Switch Applications

	Network Verification	Temporal-Uncertainty-Aware Network Verification
Number of Missed Errors (False Positives)	8	0
Number of Incorrect Alerts (False Negatives)	0	0

Table III. Errors in Error Detection by Network Verifiers, 172 BGP Edge Routers

	Network Verification	Temporal-Uncertainty-Aware Network Verification
Number of Missed Errors (False Positives)	7,098	209
Number of Incorrect Alerts (False Negatives)	137	12

conducted experiments on a small two-level fat tree network topology running learning switch applications. Both OpenFlow switches and the controllers were emulated, while the hosts were simulated. Every host was set to send traffic to any other hosts. Three sets of traces were collected from the testbed: (1) flow entries at the controller with the old network verifier, (2) flow entries (with confirmations) at the controller with the new network verifier, and (3) flow entries installed at the switches. To obtain the third set of traces, we slightly modified the OpenFlow switch code running in the VEs to output the flow entries. All three traces were then fed into the network verifier to generate error reports. Since traces from the switches represent the actual states of the network, they serve as the ground truth. The number of missed errors (false positives) and the number of incorrect alerts (false negatives) are shown in Table II. We can see that the old network verifier failed to detect eight errors out of 12 total flow entry updates (further investigation revealed that all eight errors are black-hole errors). With the uncertainty model integrated, the new design is able to capture all the transient black-hole errors in this case.

Next, we conducted experiments on a larger network scenario with the same mechanism. The network contained 172 edge routers constructed with Rocketfuel topology data [University of Washington 2002]. We fed the routers with a stream of real BGP traces (18,228 rules) [University of Oregon 2005] in our testbed and collected the three sets of traces as we did in the previous experiments; the results are shown in Table III. While both designs have missed errors and wrong alerts in this scenario, the uncertainty-aware network verifier has much higher detection accuracy than the old design. The new design was able to capture 6,889 more real errors than the old design did and experienced 125 fewer wrong alerts. With the evaluation results from our testbed, we are confident that the uncertainty-aware network verifier will outperform the old design by capturing more transient network errors when it is integrated into real systems.

8. CONCLUSION AND FUTURE WORK

This article describes how we extended our network testbed, consisting of virtual-machine-based emulation and parallel simulation, to support OpenFlow-based SDNs, and our efforts to make the testbed scalable, including a new global emulation/simulation synchronization algorithm, an asynchronous synchronization algorithm for passive simulated controllers, and a two-level architecture design for active simulated controllers. The evaluation results indicate that the testbed scales with large network model size and has better performance fidelity than Mininet. We present a case study on the use of our testbed for testing and evaluation of various designs of SDN-based applications on network verification.

We plan to evaluate the network-level and application-level behaviors for different SDN applications under various network scenarios (e.g., long delay or lossy link). We also plan to investigate techniques to extract lookahead specific to SDN models to further improve the system performance. In addition, we will utilize the testbed to design and evaluate more SDN applications, especially in the context of the smart grid; an example would be design of efficient quality-of-service mechanisms in substation routers, where all types of smart grid traffic aggregate. Currently, the testbed does not support experimentation with real SDN hardware in the loop, which is a challenging problem especially when real devices generate heavy workload. We plan to investigate means to extend the virtual time concept to certain types of real SDN switches for seamless connection of real hardware in our testbed.

ACKNOWLEDGMENTS

The authors are grateful to Ms. Wenxuan Zhou and Prof. Matthew Caesar for their ideas on uncertainty-aware data plane verification in SDN-based networks and valuable discussions on running simulation/emulation experiments for the case study.

REFERENCES

Ehab Al-Shaer, Will Marrero, Adel El-Atawy, and Khalid ElBadawi. 2009. Network configuration in a box: Towards end-to-end verification of network reachability and security. In *Proceedings of the 17th IEEE International Conference on Network Protocols (ICNP)*. 123–132.

George Apostolopoulos and Constantinos Hassapis. 2006. V-eM: A cluster of virtual machines for robust, detailed, and high-performance network emulation. In *Proceedings of the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 117–126.

Rassul Ayani. 1988. A parallel simulation scheme based on distances between objects. *Royal Institute of Technology, Department of Telecommunication Systems-Computer Systems* (1988).

Craig Bergstrom, Srinidhi Varadarajan, and Godmar Back. 2006. The distributed open network emulator: Using relativistic time for distributed scalable simulation. In *Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation (PADS)*. IEEE, 19–28.

Pratik K. Biswas, Constantin Serban, Alex Poylisher, John Lee, Siun-Chuon Mau, Ritu Chadha, Cho-Yu J. Chiang, Robert Orlando, and Kimberly Jakubowski. 2009. An integrated testbed for virtual ad hoc networks. In *Proceedings of the International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*. IEEE Computer Society, Los Alamitos, CA, 1–10.

K. Mani Chandy and Jayadev Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on Software Engineering* 5 (1979), 440–452.

Phillip M. Dickens, Philip Heidelberger, and David M. Nicol. 1994. A distributed memory LAPSE: Parallel simulation of message-passing programs. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation (PADS'94)*. ACM, New York, NY, 32–38.

Miguel A. Erazo, Yue Li, and Jason Liu. 2009. SVEET! a scalable virtualized evaluation environment for TCP. In *Proceedings of 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops (TridentCom)*. IEEE, 1–10.

Google. 2011. Inter-Datacenter WAN with centralized TE using SDN and OpenFlow. (2011). Retrieved from https://www.opennetworking.org/images/stories/downloads/misc/googlesdn.pdf.

Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. 2008. Time jails: A hybrid approach to scalable network emulation. In *Proceedings of the 22nd Workshop on Principles of Advanced and Distributed Simulation (PADS)*. IEEE, 7–14.

Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat. 2008. DieCast: Testing distributed systems with an accurate scale model. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. USENIX Association, Berkeley, CA, 407–422.

Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. 2005. To infinity and beyond: Time warped network emulation. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP'05)*. ACM, New York, NY, 1–2. DOI:http://dx.doi.org/10.1145/1095810.1118605

Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. 2012. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies*. ACM, 253–264.

Brandon Heller. 2013. *Reproducuble Network Research with High-Fidelity Emulation*. Ph.D. Dissertation. Stanford University.

Dong Jin and David M. Nicol. 2010. Fast simulation of background traffic through fair queueing networks. In *Proceedings of the 2010 Winter Simulation Conference (WSC'10)*. 2935–2946. DOI:http://dx.doi.org/10.1109/WSC.2010.5678988

Dong Jin and David M. Nicol. 2013. Parallel simulation of software defined networks. In *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 91–102.

Dong Jin, Yuhao Zheng, Huaiyu Zhu, David M. Nicol, and Lenhard Winterrowd. 2012. Virtual time integration of emulation and parallel simulation. In *Proceedings of the 2012 Workshop on Principles of Advanced and Distributed Simulation (PADS'12)*. 120–130.

Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. 2013. Real time network policy checking using header space analysis. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*. USENIX Association, Berkeley, CA, 99–112.

Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, 1.

Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying network-wide Invariants in real time. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI'13)*.

Jereme Lamps, David M. Nicol, and Matthew Caesar. 2014. TimeKeeper: A lightweight virtual time system for linux. In *Proceedings of the 2nd ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*. ACM, 179–186.

Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A network in a laptop: Rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*.

Jason Liu, Raju Rangaswami, and Ming Zhao. 2010. Model-driven network emulation with virtual time machine. In *Proceedings of the 2010 Winter Simulation Conference (WSC'10)*. IEEE, 688–696.

Boris D. Lubachevsky. 1989. Efficient distributed event-driven simulations of multiple-loop networks. *Communications of the ACM* 32, 1 (1989), 111–123.

LXC. 2013. Linux Containers. Retrieved from https://linuxcontainers.org.

Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P. Brighten Godfrey, and Samuel Talmadge King. 2011. Debugging the data plane with Anteater. In *Proceedings of the ACM SIGCOMM Computer Communication Review (SIGCOMM'11)*. ACM, New York, NY, 290–301. DOI:http://dx.doi.org/10.1145/2018436.2018470

Steffen Maier, Andreas Grau, Harald Weinschrott, and Kurt Rothermel. 2007. Scalable network emulation: A comparison of virtual routing and virtual machines. In *Proceedings of the 12th IEEE Symposium on Computers and Communications (ISCC'07)*. 395–402. DOI:http://dx.doi.org/10.1109/ISCC.2007.4381529

Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* 38, 2 (2008), 69–74.

David Nicol and Guanhua Yan. 2006. High-performance simulation of low-resolution network flows. *Journal of Simulation* 82, 1 (2006), 21–42.

David M. Nicol. 1993. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM (JACM)* 40, 2 (1993), 304–333.

David M. Nicol, D. Jin, and Y. Zheng. 2011. S3F: The scalable simulation framework revisited. In *Proceedings of the 2011 Winter Simulation Conference*. Phoenix, AZ.

David M. Nicol and Jason Liu. 2002. Composite synchronization in parallel discrete-event simulation. *IEEE Transactions on Parallel and Distributed Systems* 13, 5 (2002), 433–446.

ns-3. 2011. The ns-3 Project. Retrieved from http://www.nsnam.org.

ns-3. 2013. Link Modeling Using ns 3. Retrieved from https://github.com/mininet/mininet/wiki/Link-modeling-using-ns-3.

ns-3 OpenFlow Model. 2011. ns-3 OpenFlow Switch Support. Retrieved from http://www.nsnam.org/docs/release/3.13/models/html/openflow-switch.html.

OFTest. 2011. OFTest, a Python Based OpenFlow Switch Test Framework. Retrieved from http://www.openflow.org/wk/index.php/OFTestTutorial.

Open vSwitch. 2011. An Open Virtual Switch. Retrieved from http://openvswitch.org/.

OpenFlow. 2011. OpenFlow Switch Specification Version 1.1.0. Retrieved from http://www.openflow.org/documents/openflow-spec-v1.1.0.pdf.

OpenVZ. 2006. OpenVZ Linux Containers. Retrieved from http://wiki.openvz.org.

POX. 2011. Python OpenFlow Controller. Retrieved from http://www.noxrepo.org/pox/about-pox/.

QEMU. 2009. QEMU, Open Source Processor Emulator. Retrieved from http://wiki.qemu.org/.

Rob Sherwood. 2011. OFlops. Retrieved from http://www.openflow.org/wk/index.php/Oflops.

Rob Simmonds, Cameron Kiddle, and Brian Unger. 2002. Addressing blocking and scalability in critical channel traversing. In *Proceedings of the 16th Workshop on Parallel and Distributed Simulation (PADS'02)*. IEEE Computer Society, Washington, DC, 17–24.

SSF. 1999. SSF: Scalable Simulation Framework. Retrieved from http://www.ssfnet.org/.

Stanford University. 2009. OpenFlow Switching Reference System. Retrieved from http://www.openflow.org/wp/downloads/.

Stanford University. 2012. Mininet: An Instant Virtual Network on Your Laptop (or Other PC). Retrieved from http://mininet.org/.

UML. 2006. The User-Mode Linux Kernel. Retrieved from http://user-mode-linux.sourceforge.net.

University of Illinois at Urbana-Champaign. 2013. Ocean Cluster for Experimental Architectures in Networks (OCEAN). Retrieved from http://ocean.cs.illinois.edu/.

University of Oregon. 2005. Route Views Project. Retrieved from http://www.routeviews.org/.

University of Washington. 2002. Rocketfuel: An ISP Topology Mapping Engine. Retrieved from http://www.cs.washington.edu/research/rocketfuel/.

Virtuozzo. 2012. Parallel Virtuozzo Containers. Retrieved from http://www.parallels.com/products/pvc46.

VMware. 1998. VMware virtualization software. (1998). Retrieved from http://www.vmware.com

Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. 2002. Scale and performance in the Denali isolation kernel. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, ACM Operating Systems Review, Winter 2002 Special Issue, 195–210, Boston, MA, USA.

Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. 2002. An integrated experimental environment for distributed systems and networks. *ACM SIGOPS Operating Systems Review* 36, SI (2002), 255–270.

Xen. 2013. The Xen Project. Retrieved from http://www.xenproject.org.

Zhonge Xiao, Brian W. Unger, Rob Simmonds, and John Cleary. 1999. Scheduling critical channels in conservative parallel discrete event simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*. 20–28. DOI: http://dx.doi.org/10.1109/PADS.1999.766157

Srikanth B. Yoginath, Kalyan S. Perumalla, and Brian J. Henz. 2012. Taming wild horses: The need for virtual time-based scheduling of VMs in network simulations. In *Proceedings of the 20th International Symposium on Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 68–77.

Yuhao Zheng, Dong Jin, and David M. Nicol. 2013. Impacts of application lookahead on distributed network emulation. In *Proceedings of the 2013 Winter Simulation Conference (WSC'13)*.

Yuhao Zheng, David M. Nicol, Dong Jin, and Naoki Tanaka. 2011. A virtual time system for virtualization-based network emulation and simulation. *Journal of Simulation* 6, 3, 205–213.

Received January 2014; revised May 2015; accepted August 2015