ScalaSEM: Scalable Validation of SDN Design With Deployable Code

Nan Zhu
School of Computer Science
McGill University
nan.zhu@mail.mcgill.ca

Wenbo He
School of Computer Science
McGill University
wenbohe@cs.mcgill.ca

***Abstract*—Software Defined Networking (SDN) has been emerging to be a new paradigm of datacenter network architecture. As SDN in the datacenter environment continues to grow in scale and complexity, one of the challenges to the SDN researchers and developers is to verify a SDN design before deployment. Although there are several existing tools developed with the goal to fill this need, they are unfortunately either lacking the seamless porting ability from a simulated environment to a real deployment (*simulation-based approach*) or suffering from the scalability issue (*emulation-based approach*).**

In this paper, we propose *ScalaSEM*, a system leveraging the advantages of both simulation and emulation methods to valid the SDN design. ScalaSEM establishes a high-fidelity environment with the real-world OpenFlow-based communication channel and abstracts networks with higher yet accurate-enough level. Through two usage scenarios and the comparison with the state-of-the-art solutions, we demonstrate that *ScalaSEM* provides the validating solution to the SDN design which can scale to the network with the scale of tens of thousands hosts and thousands of machines, and imposes no necessary to modify the implementation of the SDN design to move between validation and deployment environment.

I. Introduction

Software Defined Networking (SDN) has been emerging to be a new paradigm of networking. With SDN, networks no longer require the labor-cost, error-prone and switch-to-switch configuration [18]. Instead, networks are transformed into an open and programmable component in a large cloud infrastructure, leading to more efficient and automatic network management and provisioning. Hence, SDN technologies have been adopted in a wide range of applications especially in data center environments, including load balancing, flow scheduling [2], energy-efficient network design [10], etc. According to the CRN report [16], the SDN market is expected to be valued at about USD $3.7 billion by 2016, compared to USD $360 million in 2013.

As SDN continues to grow in scale and complexity, it is important and challenging to test and verify a SDN design before the deployment for datacenter environment. The ideal validation tool is expected to have the two following features: 1) Offering the facilitates to validate SDN design targeting to the large-scale datacenter network consisting of thousands of routers and tens of thousands of servers; 2) Minimizing the gap between validation and production environment, which requires not only the high-fidelity simulation/emulation results but also the minimum overhead for porting the SDN implementation used in validation to real production environment.

Unfortunately, none of the state-of-the-art methods address both of the aforementioned challenges very well. 1) The physical-device-based testbed are less practical regarding the scalability though they have the best fidelity; 2) Emulators, like Mininet [17], model each server/switch in the network with an independent process, which is less expensive than testbed but does not scale either when the network size is very large; 3) The simulation-based approaches [9] fail to capture all possible cases which happen in real-world SDN environment as it replaces the real network component, e.g. SDN controller with the simulated one; additionally, it requires further efforts, e.g. modify the implementation with the risk of changing the program behavior, to port the SDN implementation between simulation and deployment environment.

In this paper, we propose ScalaSEM[1], a scalable approach for validating SDN design with the deployable code. To reproduce the real-world SDN behavior with the high fidelity, we implement OpenFlow protocol in ScalaSEM and conduct validation with real OpenFlow messages. Hence, the communication between the OpenFlow switches and SDN controller is *exactly the same as that in real-world deployment*. The standard-OpenFlow-based communication enables the SDN controller implementation running smoothly in ScalaSEM environment to run in real production environment directly without any modification. Instead of focusing on each packet traveling through networks, which usually brings unnecessary overhead in most of the packet-level simulators, we abstract network at flow level, so that we focus on the flow routing, rate control, etc. which are the major research problem in the published network research works [2] [10]. To further improve the scalability of ScalaSEM, the communication of the simulated/emulated network components in ScalaSEM are following by multi-threaded and asynchronous pattern, which enables us to handle the network consisting of as many as 27000+ servers and 2800+ switches.

The validation environment for ScalaSEM is shown in Figure 1. We design the OpenFlow switches in ScalaSEM with a hybrid model of emulation and simulation. Though the switches are essentially the in-memory objects of ScalaSEM process, they are equipped with an emulated OpenFlow module which communicates with the real-world OpenFlow controllers via TCP connections. The emulated OpenFlow module sends, receives and reacts to the real standard OpenFlow messages. With this design, ScalaSEM works with the real-

[1]Source code of ScalaSEM is available at https://github.com/CodingCat/scalasem.

978-1-4673-8590-9/15/$31.00 ©2015 IEEE

world OpenFlow-based control applications seamlessly. The traffic is simulated at flow level with the critical attributes, e.g. source/destination address in L2/L3, flow rate, total bytes, remaining bytes to send, etc.

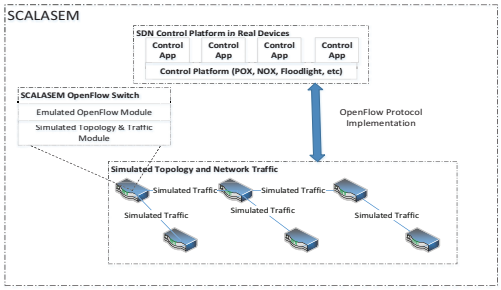


Fig. 1: ScalaSEM Environment: ScalaSEM implements OpenFlow protocol on simulated OpenFlow switches to communicate with the real-world OpenFlow control platform. In this way, ScalaSEM provides the seamless portability for the control applications migrating from ScalaSEM to testbed deployment and vice versa directly.

We evaluate ScalaSEM performance in terms of efficiency, scalability, and correctness. We compare the length of emulation/simulation duration of ScalaSEM and Mininet under the topologies with varying sizes and under different traffic patterns. It demonstrates that ScalaSEM is 200x faster than Mininet in a network with moderate size. Also, ScalaSEM is scalable to an extremely large scale network with 27000 servers. We implement Hedera [2] and DevoFlow [6] in ScalaSEM. These two use cases show that ScalaSEM provides seamless portability for control applications and enables us to evaluate the customized designs of OpenFlow switches. We compare the results generated by ScalaSEM with the ones measured by the authors of Hedera and DevoFlow papers respectively [2][6], proving that ScalaSEM reproduced the results with high fidelity.

The remainder of this paper is organized as follows. In Section II, we show the limitations of the existing solutions in simulating/emulating OpenFlow networks. We describe the design rationale of ScalaSEM in Section III and detailize the multi-threaded asynchronous OpenFlow communication module in Section IV. In Section V we show two use cases of ScalaSEM. In section VI, we compare ScalaSEM with Mininet in different experiment scales and compare the results generated by ScalaSEM with those generated in the testbed environment, proving the accuracy of ScalaSEM. We make conclusions and discuss the future work in Section VII.

II. Related Work

While the tools to validate Software Defined Network have been advancing in the past three years, they still have various limitations. In this section, we analyze the scalability, flexibility and the realism of these solutions, highlighting the points which drive the design of ScalaSEM.

Emulation-based Tools Mininet [17] is the most widely used emulator of SDN. It yields the emulation results close to the deployed system and seamless portability for the controller applications from Mininet environment to the testbed deployment and in reverse. Each host in Mininet is implemented as an independent process in the Linux network namespace, and all hosts are connected by fully functional virtual Ethernet devices. This design compromises the scalability of the tool. As shown in Section VI, Mininet can support emulated networks with at most hundreds of servers. DOT [20] distributes the emulation workload to a cluster of machines, however, it brings much more deployment complexity to the user, because it requires user to deploy the virtual machines, guest OSes as well as the virtual switches on each physical machine and additionally, install the control node for the emulation task. Hence, it still cannot scale to the size of datacenter networks.

Simulation-based Tools fs-sdn [9] is a SDN simulator which provides good scalability by adopting a higher-level abstraction called *flowlet*. It only calculates the volume of a flow emitted over a given period so as to avoid simulating the arrival and departure of every packet. *fs-sdn* is developed based on POX [5], a Python-based SDN control platform. *fs-sdn* simulates the traffic between the OpenFlow switches and the controller by modifying POX classes at runtime to eliminate the network dependency of POX. The feasibility of this approach relies on the fact that both POX and *fs-sdn* are developed in Python; in another word, *fs-sdn* cannot be used with any other controllers developed in other programming languages, e.g. NOX [8] in C++, Floodlight [4] in Java. The simulator *sts* [21] focuses on simulating the OpenFlow switch behaviour to facilitate the debugging for control applications.

The above methods fail to provide either enough scalability or seamless portability from validation environment to the deployment. The goal of ScalaSEM is to overcome these issues and offer an environment supporting the scalable validation of SDN design with the directly deployable code.

III. Design of ScalaSEM

We describe the design rationale of ScalaSEM in this section. Specifically, we introduce the architecture and the working mechanism of ScalaSEM in Section III-A; In Section III-B, we illustrate why ScalaSEM's design can capture those hard-to-find bugs in SDN controller design and why conventional simulation-based approaches fail; Finally, we describe the effective flow-level abstraction in ScalaSEM in Section III-C.

A. Overview

ScalaSEM validates SDN design by creating and replaying the discrete events in networks. These events can be the flow start/end and the interaction between the OpenFlow switches and the controllers, e.g. the controller install flow entries in the flow table of the switches or the switch inquires the controller for the action to be applied to a particular flow. The events involve simulated components like flow, links, servers, simulation module in the switch as well as the emulated ones like controller, OpenFlow module in the switch, etc. The validation is essentially to test the components' behavior under the various composition of the events.

Figure 2 shows the layered architecture of ScalaSEM:

- **Infrastructure Layer** serves as a discrete event simulation engine. The Infrastructure layer provides API for other components of ScalaSEM to generate new

events and pushes forward the validation process in ScalaSEM by consuming these events.

- **Logic Layer** consists of flow simulation, topology simulation and the switch simulation/emulation. This layer creates and updates the state of the involved components in the discrete events, e.g. the remaining amount of bytes to transfer of a particular flow will be updated in this layer with the progress of validation process. Flow simulation module abstracts the network at flow level and maintains the status of the flows, e.g. routing path, amount, acquired bandwidth, etc. Topology simulation builds the interested network topology with the simulated switches/servers/links. The essence of ScalaSEM design, the integration of emulation and simulation, is reflected by the *Switch* design. Switch Emulation contains SwitchManager which manages the connection with the controller. Although OpenFlow switches in ScalaSEM are simulated with the in-memory objects in the heap space of the program, the connection with the OpenFlow controller is the real TCP connection through which the standard OpenFlow packets are transferred just like in real-world networks. For better, we implement a Multi-threaded and Asynchronous network communication module to handle the OpenFlow connections initialized by ScalaManagers. We will detailize the design in Section IV. Switch simulation defines the functionality about the flow forwarding and the bandwidth resource allocation among the flows.

- **API Layer** provides THE interfaces for the user to reproduce/validate SDN system behaviour under different workload, topology, etc. The examples in Section V are the use cases of the APIs in this layer.

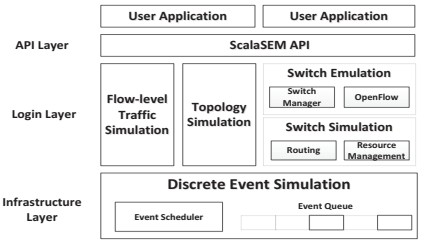


Fig. 2: Layered Architecture of ScalaSEM, consisting of Infrastructure, Logic and API Layer. ScalaSEM validates the SDN design by creating and replaying the discrete events in network.

From the architectural point of view, ScalaSEM distinguishes itself with the existing solutions in three aspects: 1) The integration of emulation and simulation design in ScalaSEM brings good scalability to handle large scale network while keep the trustable fidelity; 2) The communication between the OpenFlow switches and the controller is through real OpenFlow messages, enabling more fidelity validation process and seamless porting ability from validation to deployment environment; 3) Multi-threaded and asynchronous communication module further improves the performance in terms of scalability.

B. OpenFlow-based Communication

In this section, we explain why we implement OpenFlow protocol in ScalaSEM instead of converting OpenFlow-based interaction between the switches and the controller into the function calls in the simulator.

Though the simulation-based approaches are lightweight and scalable, they usually do not deal with the implementation details on real world devices, thus the simulation results are less trustable than emulation-based approaches. In a SDN design, since control application plays a critical role in the network performance, the pure simulation approach may not capture the subtlety in control policies, hereby may not yield trustable results.

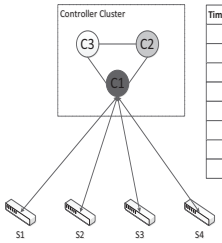
One of the examples is that the simulation-based approach like *fs-sdn* ignores the network dependency in control applications, and replaces it with sequential function calls. However, in real world, the controller usually starts concurrent threads to handle concurrent OpenFlow messages sent from different switches, so that the sequential functions in simulation-based approaches cannot test the concurrency control policy in the implementation of control applications.

Figure 3 illustrates a typical case in which a simulation-based approach can not accurately reproduce the OpenFlow controller behavior in the real world. The presented architecture is exactly the same as the distributed controller architecture in Floodlight [4], a mature SDN control platform product. In the example, multiple controllers (C1, C2, C3) coexist in an OpenFlow network to eliminate the single point of failures. Each controller is multi-threaded for higher throughput. These controllers are organized in a master-slaves manner, i.e. one of the controllers is actively managing all the connected switches while others are running as the backups. Certain election algorithm is applied to ensure that one of the backups (C2, C3) is taking the role of the original active master (C1) once C1 is out of service. Both the old and the newly elected master send "RoleChange" message to the switches to direct them to initialize the new connections to the new master and discard the connection with the old one.

Under the scenario of Figure 3, some of switches are likely to be out of service without the careful SDN controller design. Suppose a newly connected switch (S4) is handshaking with the old master controller (C1) with OpenFlow messages just at the moment when the new master is elected. In this case, C1 starts a thread (T1) to handle the handshake request. Before T1 is scheduled, C1 receives the notification that C2 is the new master controller. Upon receiving the notification, C1 starts thread T2 to notify the already-connected switches S1, S2 and S3 to switch to C2. If T2 is scheduled before T1, S4 cannot successfully connected to C2 since it is not taken as the "already-connected" switch by C1 before C1 sends the notification about the new master. The complete process is detailed as timestamp 1 - 8 in the Figure. The figure illustrate a race condition bug existing on the controller side. To resolve the bug, we need to ensure that T1 always runs before T2.

The simulation-based approaches cannot capture this bug, because the simulators converts the communication between the switches and the controllers converts into sequential function calls. With simulation-based approach, the action happened in timestamp 5 and 8 in Figure 3 would always be an

atomic process (executed in the same function call), thus the issue described above will never happen in a SDN simulator. In summary, it is hard for researchers and developers to catch the bug and correctly validate their controller design with the simulation-based approaches. With ScalaSEM, multiple switches send OpenFlow messages to the controller concurrently, and the controllers run in the same way as they are in deployment envrionment. Thus the race condition bug existing in controller would be exposed eventually.



Timestamp	Object	Action
1	C1	C1 is the master controller
2	C2,C3	C2, C3 run as slave controllers
3	S1,S2,S3	S1, S2, S3 connects with C1
4	S4	S4 sends handshaking with C1 to join network
5	C1	C1 starts new thread T1 to handle message sent in 4
6	C1,C2,C3	C1 gets lost with C2, C3, C2 is elected as new master
7	C1	C1 is recovered and aware of new master, it starts a new thread T2 to notify the switches about the new master
8	C1	T1 in C1 add S4 as new connected switch

Fig. 3: An example of race condition at controller side: newly joined switch S4 will get lost with the new master if the election of the master controller occurs at the same time with his joining.

C. Flow-level Traffic Simulation

Despite the advantages of the emulation-based approaches, scalability is the major concern on emulation, as we stated in Section II.

To simulate/emulate the real network behaviour, the existing approaches usually abstract the network behaviour in either packet or flow level. At the packet level, the simulator has to capture the arrival and departure of every packet on each network device (hosts and routers/switches). Given the MTU (Maximum Transmission Unit) as 1500 bytes, to simulate the 1Mbps traffic on a single router, we need to generate more than 1200 discrete events in every second in virtual time, which is obviously an intolerable overhead when simulating a typical datacenter network containing thousands of routers/switches and hosts.

In ScalaSEM, we simulate the network behavior at *flow level*. 1) Instead of focusing on the arrival and departure of every packet, tracking the path and the volume of a flow enables us to only keep the relevant information of the flows in the in-memory data structure and update them when necessary, which is much more lightweight than capturing each packet; 2) OpenFlow [7] itself is a flow-level network management protocol, which suggests that the per-packet information does not bring much impact when we validate the OpenFlow network design.

When we create a new flow in the simulated network topology, we divide the process into two phases: *Routing* and *Resource Allocation*. ScalaSEM first decides the path for each flow by querying the controller or forward according to the flow table entry on each switch (Routing). In Resource Allocation, we traverse each link along the path and allocates the bandwidth according to the rules specified by the controller. By default, ScalaSEM simulates the max-min fair sharing rule in TCP.

The issue in flow-level simulation is "ripple effect". In the "ripple effect", the status update of a certain flow triggers the status update of other flows. Hence, a large amount of events are generated and processed in the simulator. For example, allocate the bandwidth to a new flow in certain link requires the update of allocated bandwidth of all flows passing through the same link. According to max-min fair sharing rule, this update has to be propogated to the other flows and link. We cannot eliminate this case in a flow-level simulator but the results in section VI show that ScalaSEM tolerates "ripple effect" well and keeps good performance even the paths of many flows are overlapped.

IV. Parallel OpenFlow Emulation

In this section, we discuss how we apply the concurrent design in OpenFlow module in ScalaSEM and how we overcome the challenges brought by this concurrent design.

A. Asynchronous and Multi-Threaded Communication

ScalaSEM maintains a thread pool to handle the connection request sent from the SwitchManagers in Figure 2, i.e. ScalaSEM concurrently establishes connections between OpenFlow switches and the controller. The benefit of this design is two-folded: 1) In the real world, OpenFlow controllers do handle concurrent connections from multiple switches at the same time. With the concurrent connection functionality, we can closely reproduce the behavior of the OpenFlow controller thus facilitates to find problems in SDN design. 2) Multi-threaded connection handler accelerates the validation process.

To further improve the performance of ScalaSEM, the establishment and I/O operation in the connections in ScalaSEM are all asynchronous. Asynchronous design maximumly utilizes the CPU cycles to push forward the validation. When one of the switches invokes connection request to the controller, the thread performing the operation does not need to block to wait for the response from controller, instead, it returns immediately to perform other tasks, e.g. handling the response returned to another switch.

B. Preventing Partial Execution of Events

Though the aforementioned multi-threaded design boost the performance, the correctness of ScalaSEM would be influenced without the careful consideration in the implementation. We refer to this problem as "Partial Execution of Events".

As shown in the top half of Figure 4, Flow A and B sharing the same path start at the 0th second and 10th second respectively. Straightforwardly, Flow A should use the bandwidth of the links exclusively until B starts. The ingress switch sends two $PACKET\_IN$ messages to the controller when the flows arrive [2]. Two concurrent threads are started to handle two $PACKET\_OUT$ returned from the controller. Recalling that each flow in ScalaSEM decides the path at first before the bandwidth allocation, the thread handling the

[2] $PACKET\_IN$ and $PACKET\_OUT$ are two messages defined in OpenFlow protocol, $PACKET\_IN$ is sent from the switch to the controller when a new flow arrives, it contains the flow identification for the controller to generate the proper action to be applied to the flow; $PACKET\_OUT$ is sent from the controller to the switch to indicate the port from which the flow should be forwarded.

$PACKET\_OUT$ for flow A may be interrupted by the thread of flow B before flow A's bandwidth is allocated. If the thread for flow B returns after the bandwidth allocation finishes, the bandwidth would have been "contaminated" by flow B when A's thread is scheduled by the OS again.

We resolve the issue by making the two stages of a flow, *Routing* and *Resource Allocation*, as an atomic process. In the example, the "PACKET_IN" message of flow B is not sent until flow A gets its initial bandwidth. The means, the flows without the overlapped path can be handled concurrently while atomic process is applied for the correctness when there is shared resource.

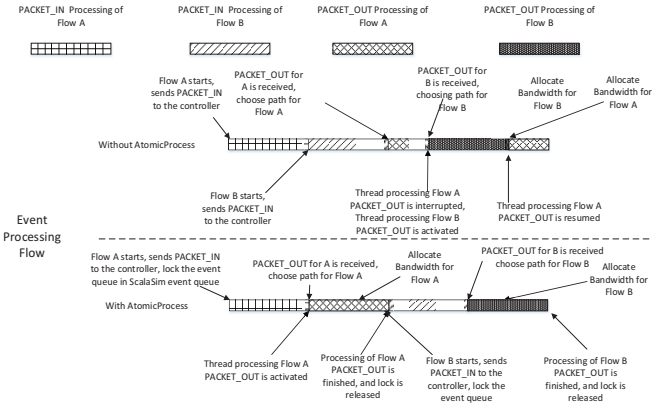


Fig. 4: Split of Event Processing: In the top half of the graph, thread handling the processing of Flow A is interrupted and the thread processing Flow B is scheduled, so the event involved Flow A is partially executed; In the bottom half, the events will not be partially executed since routing and resource allocation are organized as an atomic process.

V. Case Study

In this section, we study two usage cases to illustrate how ScalaSEM validates the design of OpenFlow controller as well as customized OpenFlow switch design. The first scenario is to show how we validate the design of a Software Defined "centralized routing controller". The centralized controllers [13] [22] [2] take the global network info as the input and usually makes the more effective routing decision than the distributed routing control plane (e.g. the conventional routing schema). In the first scenario, we choose to implement Hedera [2], an OpenFlow control application dynamically scheduling large flows to minimize the collisions in bandwidth allocation, in ScalaSEM. The second scenario is to simulate show how we validate the customized switch design in ScalaSEM. The sophisticated switch design [6] [19] [11] plays important role in SDN. In this scenario, we implement DevoFlow [6], which introduces "Local Actions" in OpenFlow switches to devolve the workload for the controllers. To prove the correctness of ScalaSEM, we compared the result from ScalaSEM and that measured in the original papers.

A. Validate Centralized Routing Controller in ScalaSEM

Hedera [2] was proposed to solve the flow collisions caused by the default Multi-Path routing mechanism in today's datacenter networks. Hedera utilizes the global OpenFlow controller to estimate the bandwidth demand of each flow, and replace the flows with the demand larger than the preset threshold to a near-optimal or optimal path to avoid the collisions.

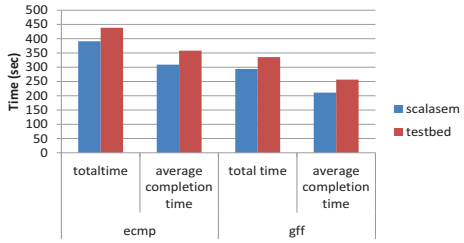


Fig. 5: Comparison between ScalaSEM and Testbed Results of Hedera: the total finish time of shuffle and average completion time of flows in ScalaSEM and Testbed are close.

ECMP (Equal Cost Multi Path) [12] is a random-hash-based strategy executed in the switches where the packet header of the flow is hashed and mapped to one of the possible outgoing paths. This strategy may forward two flows to the same link causing the collision in bandwidth allocation, thus fails to maximize the throughput even though the bisection bandwidth does exist. Hedera utilizes the global OpenFlow controller to estimate the bandwidth demand of each flow, and the flow which has the demand larger than the preset threshold is replaced to a near-optimal or optimal path to avoid the collisions with others.

To verify the accuracy of ScalaSEM-emulated Hedera, we implemented the Hedera controller based on POX [5], and used the same topology in the original paper. The Hedera authors compared the total finish time and the average flow completion time of all flows with ECMP and Hedera algorithm (Global-First-Fit) under Shuffle, a many-to-many traffic pattern. We followed the same traffic pattern and evaluate with the same metric. Figure 5 shows that the measured metrics in ScalaSEM are close to those measured in the paper [2]. In addition, when we compared the total finish and average completion time in ECMP and GFF, we found that GFF improves the performance by 31% in average finish time and 24% in total finish time, which is consistent with the value from testbed experiments, 28% in average and 23% in total. The result is shown in Table I.

	ScalaSEM	Testbed
Total Completion Time	24%	23%
Average Completion Time	31%	28%

TABLE I: Comparison of Hedera speedup under ScalaSEM and testbed (with GFF algorithm): we can see that the speedup generated by ScalaSEM-emulated Hedera is close to that in physical testbed.

B. Validate Switch Design in ScalaSEM

The centralized OpenFlow controller is a potential performance bottleneck in OpenFlow-based network. DevoFlow [6] devolves the flow management workload from the controller to the switches. One of the most important strategies adopted by

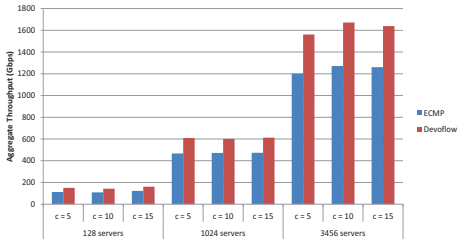


Fig. 6: Comparison between ScalaSEM-emulated DevoFlow and Original Results: the DevoFlow emulated with ScalaSEM improves the throughput by 30% and the change of concurrent connection number does not lead to the change the throughput, both of which are consistent with the original paper.

DevoFlow is "Local Action", which refers to that a switch can make decision for a certain flow by itself instead of inquiring the controller.

These actions do not impose overhead on the controller side. One of the typical local actions is the Multi-Path support. Existing mutlipath routing protocols, such as ECMP [12], focuses on using equal cost multipath for load balancing. However, the "equal cost" assumption does not always hold due to the dynamic workload. DevoFlow solves this problem by selecting an output port for a microflow according to a certain probability distribution, which is called *Oblivious Routing* [15].

We emulated DevoFlow switch with ScalaSEM to forward the flows with oblivious routing algorithm [15]. In the comparison, we used the Shuffle traffic pattern. We compared the aggregate throughput of flows with DevoFlow and ECMP. We can see that the DevoFlow improves the throughput by 30%, when the concurrent connection number is changed. Meanwhile, the throughput did not change. Both of the measured metrics in ScalaSEM are consistent with the measurement in the original paper. The result is shown in Figure 6, c is the number of simultaneous connection/server number.

VI. Evaluation

In this section, we evaluate the performance of ScalaSEM in terms of scalability, correctness, and efficiency. We describe the setup of the experiments in Section VI-A. After that, we test the scalability of ScalaSEM with increasing amount of workload and under different traffic patterns in Section VI-B. Finally, we go through the results which were presented in Section V to show the fidelity of the results generated by ScalaSEM.

A. Evaluation Setup

The baseline of the experiments in this section is Mininet, one of the most popular emulation tools. In our experiments, ScalaSEM and Mininet were run on a desktop with Intel(R) Core(TM) i3-2120 CPU @ 3.30GHz, 4 cores and 4 GB memory and the controller was run on a server with Intel(R) Core(TM) i3-3220 CPU @ 3.30GHz, 4 cores and 16 GB memory. The evaluation results shown in this section were generated under a typical fat-tree topology [1]. The topology consists of k pods (the larger the k, the more dense the network), each with two layers: edge switches (lower layer) and aggregation switches (upper layer), each layer contains $k/2$ switches. Each edge switch is attached with $k/2$ hosts. The pods are interconnected by $(k/2)^2$ core switches. Table II shows the server and switch numbers with different k. Figure 7 shows the topology when $k = 4$. The capacity of all the links is 1Gbps. For the experiment with $k = 48$, we ran ScalaSEM in the server with Intel(R) Core(TM) i3-3220 CPU @ 3.30GHz, 4 cores and 16 GB memory.

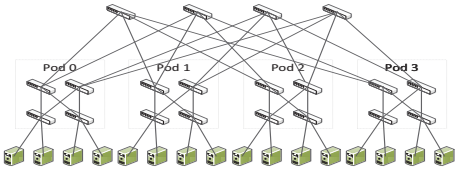


Fig. 7: Example Topology when k = 4

switch degree (k)	server number	switch number
4	16	20
8	128	80
16	1024	320
24	3456	720
48	27648	2880

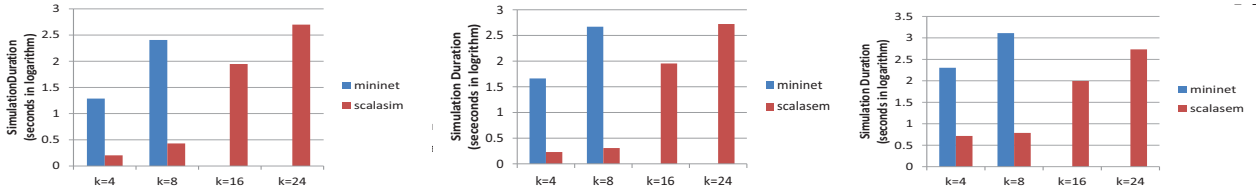
TABLE II: Simulated Network Scale: this table shows the machine and switch number against different k in the topology we simulated in ScalaSEM

B. Scalability

We first compared the performance of Mininet and ScalaSEM in terms of simulation speed, which is a good measurement of scalability. We used **Permutation Matrix** to model end-to-end traffic pattern, where each host sends a flow to a single destination chosen uniformly at random. In our experiments, after we increased k to 16, i.e. the network contains 1024 servers and 320 switches, Mininet spent more than 6 hours to create the network and no packets can be forwarded to the right location afterward. We believe that the limited scalability was caused by the creation of a large number of virtual Ethernet (veth) device pairs in Mininet and virtual bridges in OpenvSwitch. *As a result, in the figures of this subsection, we did not show the simulation duration of Mininet for k = 16 and 24 due to the crashing of the Mininet.*

From Figure 8, we can see that in Mininet, scalability declined when the network size increased. When k was 4 and 8, ScalaSEM performed at least 19x better than Mininet, because in ScalaSEM the hosts and switches are implemented as in-memory objects while Mininet has to create veth pairs bringing much overhead. Even when the network size was as large as $k = 24$, i.e. 3456 servers and 720 switches, ScalaSEM finished the emulation within 9 minutes. The Y axis in Figure 8 is drawn in logarithm for comparison convenience. We show the absolute value of simulation duration in Table III.

One interesting observation in our experiments is that, when we increased k to 24, *POX cannot handle so many simultaneous connections*, this bottleneck remains even we changed the POX source code to use *epoll* instead of default *select* to handle network connections. ScalaSEM scaled beyond POX capacity easily. To simulate extremely large-scale



(a) PermuMatrix 100MB (b) PermuMatrix 200MB (c) PermuMatrix 1000MB

Fig. 8: The simulation duration of Mininet and ScalaSEM under simulated network with different sizes and flow sizes.

scenario based on POX, we slowed down the connection speed sent to POX at the cost of extended simulation duration.

Figure 9 shows the sensitivity of Mininet and ScalaSEM to the flow size. We found that the change on the simulation duration of ScalaSEM is ignorable when we increased the flow size from 100MB to 200MB and then to 1000MB, however, the performance of Mininet degraded a lot with the increasing of flow size. The reason is that Mininet generates real packets for each flow while ScalaSEM adopts a flow-level abstraction so that the large flows do not impose much overhead to ScalaSEM.

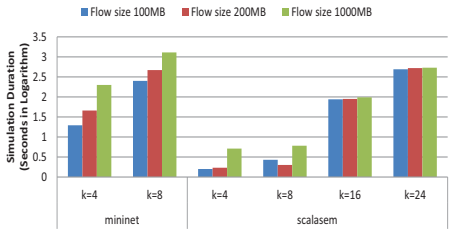


Fig. 9: Simulation Time Change of ScalaSEM and Mininet under Different Flow Size (Permutation Matrix): the performance of Mininet degrades a lot with the increasing of flow size.

k	ScalaSEM			Mininet		
	100	200	1000	100	200	1000
4	1.6	1.7	5.2	19.4	46	201
8	2.7	2.029	6.12	254.7	468	1289
16	88.606	90	99	N/A	N/A	N/A
24	500.087	528	540	N/A	N/A	N/A

TABLE III: Simulation duration of ScalaSEM and Mininet under different topology size and flow size (In MB).

Shuffle Shuffle is usually generated by distributed computing system, e.g. Hadoop, to transfer data between computing phases. From Figure 10, we can see that the performance of Mininet degraded further under shuffle, it took as long as 23 minutes to finish when $k = 8$, comparing to 4.2 minutes in permutation matrix. ScalaSEM performed much better, it takes only 2 minutes in Shuffle when $k = 8$ and only 25 minutes in a large network when $k = 24$.

Broadcast Next, we evaluated ScalaSEM and Mininet with Broadcast traffic patterns. As we stated above, the fluid-model-based ScalaSEM may suffer from the "ripple effect". In our Broadcast pattern, a single machine can send as many as 10 simultaneous flows to others, i.e. the rate change of a single

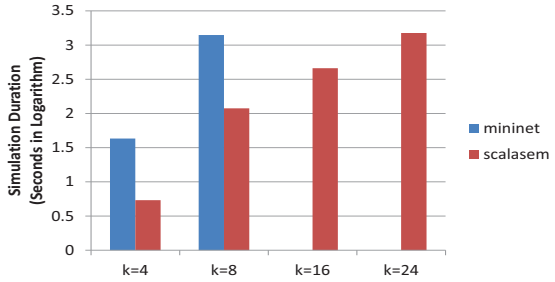


Fig. 10: Simulation Time of ScalaSEM and Mininet on logarithmic scale under Shuffle. The simulation time for Mininet and ScalaSEM are 43 seconds and 5.4 seconds respectively when $k = 4$; and are 1404 seconds and 119 seconds respectively when $k = 8$. When $k = 16$ and $k = 24$, Mininet crashes, while the simulation time for ScalaSEM is 459 and 1504 seconds, respectively.

flow can lead to the status change of at least 9 other flows, bringing more chances of happening of "ripple effect".

Broadcast traffic is usually generated by the distributed computing jobs. Researchers from Berkeley analyzed the trace of production Hadoop cluster and stated that as many as 87% jobs in the production cluster has no more than 10 tasks [3]. To be close to the real-world case, we started "broadcast" jobs, each of which generated flows from 1 to 10 servers, and we adapted the number of jobs according to the size of simulated network. Table IV describes the broadcast job number and arrival interval in the simulated topology with different sizes. In each broadcast job, the sender sent 1GB data to each receiver.

switch degree (k)	broadcast job number	arrival interval (sec)
4	10	1
8	100	1
16	500	1
24	1000	1

TABLE IV: Broadcast job number and arrival pattern under different network size

As shown in Figure 11, Mininet still works with limited scalability and it crashed after the simulated topology was with 1024 servers and 320 switches. When $k = 8$, Mininet had to spend 5 minutes to finish all jobs while ScalaSEM took as short as 31 seconds. Comparing the simulation duration of ScalaSEM under broadcast and permutation matrix in Figure 8c, we found that the "ripple effect" indeed introduced negative

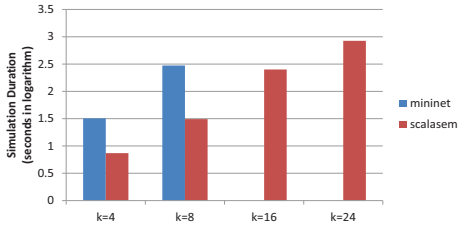


Fig. 11: Simulation Time of ScalaSEM and Mininet on logarithmic scale under Broadcast. The simulation time for Mininet is 32 seconds, for ScalaSEM is 7.4 seconds when $k = 4$, and are 298 seconds and 31 seconds respectively when $k = 8$. When $k = 16$ and 24, Mininet crashes, while ScalaSEM spends 252 and 842 seconds respectively.

influence to the simulation speed, e.g. when $k = 24$, ScalaSEM took 540 seconds to finish permutation matrix but 842 seconds to finish broadcast. However, the length of the simulation duration is acceptable considering the limited scalability and performance of the other solutions.

Extreme Large Permutation Matrix In the case when the developers or researchers need to test the controller behavior or new switch design in an extremely large scale, the validation platform like ScalaSEM is expected to support the extreme large scale simulation/emulation. To evaluate ScalaSEM under this scenario, we simulated the topology with $k = 48$ (including 27468 servers) and generated permutation matrix traffic. ScalaSEM takes 4 hours to finish all flows. When $k = 48$, we had to slow down the connection speed to 1 connection per second, otherwise POX would close some connections to the ScalaSEM switches for the reason stated earlier.

C. Correctness

As shown in Section V, Figure 5 6 represent the results generated by ScalaSEM for Hedera and DevoFlow, respectively. Both results are close to the measurements from the original papers. Since the routines of Hedera and DevoFlow cover different flow conditions and deal with different patterns of controller-and-switch interactions, we believe that the experiments with ScalaSEM will generate trustable results in general.

VII. Conclusion

In this paper, we introduced ScalaSEM, a system providing scalable solution to validate the SDN design before the real deployment. Additionally, ScalaSEM eliminates the necessary to modify the controller implementation to fit to the simulation environment, thus, we can use *deployable* code to validate the design without the risk of changing the behavior of the program expectedly during the test. Comparing to the existing solutions, ScalaSEM emulates the communication between the switches and the SDN controller with real OpenFlow messages while providing the high-level network abstraction to achieve both scalability and high fidelity.

In future work, we plan to improve the usability of ScalaSEM by providing more reusable components and popularize ScalaSEM as a widely accepted tool for SDN research.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM.

[2] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: dynamic flow scheduling for data center networks. NSDI'10, pages 19–19, Berkeley, CA, USA, 2010. USENIX Association.

[3] G. Ananthanarayanan, A. Ghodsi, S. Shenker, and I. Stoica. Effective straggler mitigation: Attack of the clones. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 185–198, Berkeley, CA, USA, 2013. USENIX Association.

[4] BigSwitch. Floodlight. http://tinyurl.com/lmwqky2.

[5] P. Community. Pox. http://www.noxrepo.org/pox/about-pox/.

[6] A. R. Curtis, J. C. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. Devoflow: scaling flow management for high-performance networks. SIGCOMM '11, pages 254–265, New York, NY, USA, 2011. ACM.

[7] O. N. Foundation. Openflow switch specification version 1.0.0. http://tinyurl.com/lb33s2o.

[8] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. Nox: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, July 2008.

[9] M. Gupta, J. Sommers, and P. Barford. Fast, accurate simulation for sdn prototyping. In *HotSDN '13*, pages 31–36, New York, NY, USA, 2013. ACM.

[10] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown. Elastictree: saving energy in data center networks. In *NSDI'10*, pages 17–17, Berkeley, CA, USA, 2010. USENIX Association.

[11] M. Honda, F. Huici, G. Lettieri, and L. Rizzo. mswitch: A highly-scalable, modular software switch. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 1:1–1:13, New York, NY, USA, 2015. ACM.

[12] C. Hopps. Analysis of an equal-cost multi-path algorithm, 2000.

[13] S. Hu, K. Chen, H. Wu, W. Bai, C. Lan, H. Wang, H. Zhao, and C. Guo. Explicit path control in commodity data centers: Design and applications. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 15–28, Berkeley, CA, USA, 2015. USENIX Association.

[14] D. Y. Huang, K. Yocum, and A. C. Snoeren. High-fidelity switch models for software-defined network emulation. HotSDN '13, pages 43–48, New York, NY, USA, 2013. ACM.

[15] M. Kodialam, T. V. Lakshman, and S. Sengupta. Traffic-oblivious routing in the hose model. *IEEE/ACM Trans. Netw.*, 19(3):774–787, June 2011.

[16] J. F. Kovar. Software-defined data centers: Should you jump on the bandwagon? http://tinyurl.com/o2448zw.

[17] B. Lantz, B. Heller, and N. McKeown. A network in a laptop: rapid prototyping for software-defined networks. Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[18] D. A. Maltz, G. Xie, J. Zhan, H. Zhang, G. Hjálmtýsson, and A. Greenberg. Routing design in operational networks: a look from the inside. In *Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '04, pages 27–40, New York, NY, USA, 2004. ACM.

[19] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 117–130, Oakland, CA, May 2015. USENIX Association.

[20] A. R. Roy, M. F. Bari, M. F. Zhani, R. Ahmed, and R. Boutaba. Design and management of dot: A distributed openflow testbed. In *Network Operations and Management Symposium*, NOMS, pages 1 – 9, 2014.

[21] S. Team. Sdn troubleshooting simulator. http://ucb-sts.github.io/sts/.

[22] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central control over distributed routing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 43–56, New York, NY, USA, 2015. ACM.