Switch-Visor: Towards Infrastructure-Level Virtualization of SDN Switches

Huan Chen
University of Electronic Science and Technology of China and Duke University

Theophilus Benson
Brown University

ABSTRACT

To test and update switch operating systems, developers and testers need to install run beta-switch OSes (switch agents) alongside production versions. However, today's network virtualization solutions fail to support infrastructure-level virtualization of hardware switches. In particular, they fail to provide performance guarantee and isolation of the switch's resources: CPU, Memory, and ASIC (TCAM/SRAM).

In this paper, we define the notion of infrastructure-level switch virtualization, akin to IaaS, infrastructure-level switch virtualization provides tenants, testers or developers, with low-level control over the switches: allowing a tenant to install switch agents on the switches and to run their own controller. To support this abstraction, we present a system, Switch-Visor, which presents a first step towards providing comprehensive virtualization of a switch's resources. Switch-Visor employs a synthesis of well-founded virtualization technologies and novel hardware virtualization techniques. Switch-Visor introduces three main concepts: first, using container-based virtualization on the switch to virtualize CPU and Memory; second, leveraging intelligent TCAM management and novel schedulers to provide guarantees within the ASIC, and employing novel domain-specific offloading techniques to eliminate sources of interference. Our proposed solutions, leverage changes to switch OS and switch agents making them immediately applicable to existing SDN switches.

CCS CONCEPTS

• **Networks → Bridges and switches**; **Network resources allocation**; **Network management**;

KEYWORDS

Software-defined Networking; Virtualization; Resource Allocation

ACM Reference Format:
Huan Chen and Theophilus Benson. 2017. Switch-Visor: Towards Infrastructure-Level Virtualization of SDN Switches. In *CAN'17: CAN'17: Cloud-Assisted Networking Workshop, December 12, 2017, Incheon, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3155921.3158431

CAN'17, December 12, 2017, Incheon, Republic of Korea

ACM ISBN 978-1-4503-5423-3/17/12…$15.00
https://doi.org/10.1145/3155921.3158431

1 INTRODUCTION

Network virtualization, the "killer-application" for Software Defined Networks (SDNs), allows network operators to carve out their networks for testing and updates. Today, network virtualization focuses on controller level virtualization [2, 15] with the virtualization layer interposed between the controller and the switches. However, with the growing adoption of white box switches, there is a growing need for a lower level virtualization primitive that enables testers and developers to install and control Switch Agents or Switch OSes for testing, updates, and outsource management functions.

This lower layer virtualization level should exist between the switch hardware and the switch agents (switch OSes) running atop it. Akin to IaaS which provides low-level control over computing, the lower layer (infrastructure-level) virtualization provides low-level control over a switch allowing a tenant to install switch agents on the switch and to run their own controller interacting with these agents. We call this new virtualization paradigm Infrastructure-level Switching as a Service (ISaaS). We argue that to provide ISaaS, SDN networks must include mechanisms and primitives to slice SDN switches across all resources: CPU, Memory, and ASIC (the switch hardware responsible for servicing SDN control plane operations (SDNCTLs)), and more importantly to provide guarantees and isolation across these resources.

Unfortunately, the complexity and heterogeneity of data plane devices, e.g. switches, makes it challenging to enforce performance guarantees on SDNCTLs, e.g. rule installation or port configuration. In particular, the response time of a SDNCTLs is a function of the switch's CPU for processing SDNCTLs and switch's ASIC hardware for implementing SDNCTLs. Furthermore, the resource requirements of these SDNCTLs vary significantly — Table 1 highlights the heterogeneity in resource requirements across several representative SDNCTLs [16]. Making performance guarantees over these SDNCTLs requires:

- Intelligently rate limiting the set of SDNCTLs contending for hardware resources.
- Controlling the set of background processes utilizing the switch's resources to eliminate hardware interference – a key challenge in making guarantees in any virtualized scenario [14, 21, 22].
- Scheduling the different SDNCTLs to simultaneously ensure that performance guarantees are met while maintaining work-conservation to eliminate resource waste.

1.1 Related Work

Traditional approaches for virtualizing SDN switches, FlowVisor [15] and OpenVirtex [2], focus on partitioning TCAM space and enforcing access control over traffic. These approaches do not make guarantees on the speed with which the SDNCTLs may be installed or

Request (1/s)	F.Mod	Flow Stats	Port Stats	Feature	Echo
5000	60%	30%	50%	42%	22%
10000	80%	48%	75%	60%	23%
20000	N/A [1]	52%	92%	85%	25%

Table 1: CPU Utilization of Different SDNCTLs

configured. On the other hand, providing performance guarantees over traffic flowing through a virtual network, switch backplane, is significantly simpler than providing guarantees for SDNCTLs performed on the virtual network, because existing data plane primitives lend themselves, more naturally, to enforcing performance isolation between traffic. Specifically, existing tools, e.g. token buckets, provide guarantee and isolation by limiting the quantity of traffic for each tenant. However, as discussed earlier it is insufficient to simply rate limit the quantity of SDNCTLs because different SDNCTLs require different amount of resources and therefore limits must be carefully determined based on the type of SDNCTLs currently being processed.

1.2 Switch-Visor

In this paper, we argue that rather than focusing on redesigning switch hardware, which has significant cost implications, we should focus on judiciously allocating and managing existing switch hardware resources to ensure that performance guarantees can be efficiently enforced. Moreover, for uncontrollable events, i.e., hardware interrupts, we should offload their processing from switches to cheaper commodity servers. Switch-Visor takes the first step towards developing a framework that provides guarantees on control plane I/O operations, SDNCTLs by introducing primitives to control ASIC-driven I/O events and mechanisms to use existing black-box models to rate limit controller I/O events for different tenants. Our design of Switch-Visor introduces three novel components to the SDN ecosystem:

Containerized Switches (CPU/Memory Virtualization): We argue for a light virtualization of a physical switch by running containers, one for each tenant. The containers allow us to isolate the different tenants and provide CPU and memory guarantees. Additionally, the containers allow a tenant to run arbitrary agents.

SDNCTL-Visor (ASIC Virtualization): To provide guarantees at the ASIC level, we propose a shim layer between the containers and the ASIC. This shim layer uses a token-bucket to rate limit interactions with the ASIC. The size of the bucket is a function of the guarantees, the current state of the switches, and the set of permitable SDNCTLs. The token buckets for the different containers on a switch are constantly updated after each SDNCTL is permitted into the ASIC. To calculate the actual rate limits with multiple resource allocation, we use Dominant Resource Fairness (DRF) [9] which is a generalization max-min fairness for multiple resources.

Event-Offloader: To handle unpredictable SDNCTLs generated by the switches, Switch-Visor offloads processing from the switch to a container running on the cloud servers. This shifts the burden of handling and managing these SDNCTLs to an x86 server where they can be rate limited (by the server's NIC).

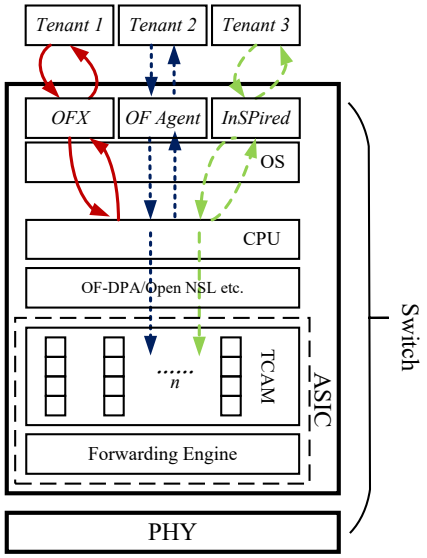


Figure 1: White Box Switch

2 BACKGROUND

Although much work has gone into host virtualization and creating solid techniques for ensuring and enforcing isolation between tenants on a host, the network (switch) has seen relatively little innovation in this direction.

In this section, we review the design and architecture of modern SDN-switches and highlight the challenges of virtualizing and sharing these switches.

2.1 White Box Switches

Most modern SDN switches are essentially white box switches running commodity software and composed of commodity switching hardware. These switches contain traditional CPU and memory structures making them amenable to existing virtualization techniques for isolating, virtualizing, and sharing the computing resources. Additionally, they contain specialized (Application-specific integrated circuit) ASICs which are used for network specific functionality – forwarding packets. Techniques for CPU and memory virtualization do not naturally lend themselves to ASIC virtualization.

Challenge 1: Switch virtualization requires tackling the ASIC.

These devices run a Linux based OS with specialized device drivers to handle interactions with the switch's ASIC. This simple and open ecosystem has fostered growth and innovation of white box switches. The openness and ability to run arbitrary software on these switches have motivated others to design new switch agents to delegate and run next to the traditional SDN agents. Moreover, a growing number of switch agents exist to run on white box switches, e.g. Aristas EoS, BigSwitch's SwitchLight, and

Switch-Visor classification	SDNCTL Type	OpenFlow SDNCTL Examples
Tenant Initiated	Controller-to-Switch	Read-State, Flow-Mod, Port-Desc Modify-State, Configuration, Features Packet-Out, Barrier
	Symmetric	Hello, Echo, Vendor
Hardware Initiated	Asynchronous	Packet-In, Flow-Removed, Port-Status, Error

Table 2: Classification of OpenFlow Messages

VMWare's OpenVSwitch (OVS). These agents will compete and content with the SDN agents for CPUs [4, 17], and hardware resources (e.g. TCAMs [4]).

Challenge 2: Tenants are able to bring their own Switch-software agent, forcing us to revisit assumptions and architectures for the switch's OS.

2.2 Virtualization Challenges

We argue that ASIC virtualization should provide guarantees over the amount and number of instructions that can be performed. Akin to the IOPS (Input/Output Operations per second) we expect for virtual Disks, virtualized ASICs should provide a similar notion of SDNCTL Operations per section (SDNCTL-OPS).

To better understand the challenges that complicate ASIC virtualization, in Figure 1, we present the diagram of white box switch and highlight components used to perform different SDNCTL operations. We roughly classify the SDNTCL into several categories and highlight related challenges [1]:

Tenant Initiated: This class of SDNCTL are generated by the controller or the local switch agent. For example, *flow-mod*, an SDNCTL used to add flow entries to the ASIC. These SDNCTLs require a combination of CPU and ASIC processing, however, these events can be measured by benchmarking the devices and controlled by rate-limiting the controller or the local switch agent. The key challenge lies in understanding how to set the weight for the rate-limiter and more specific understanding how to translate the benchmarking results into rate limits.

Hardware Initiated: These SDNCTL are akin to hardware interrupts, they are generated by the ASIC and sent to the OS, the switch agent and subsequently to the remote controller for processing. Examples include, *packet-in* which is generated when a packet arrives but the ASIC contains no flow tables. This class of SDNCTLs are harder to control because there are no rate-limiters within the hardware to suppress these events. Although uncontrollable, these SDNCTLs can significantly impact CPU performance, with prior studies showing that SDNCTL can consume significant CPU and memory resources thus significantly impacting the performance of the switch agents [7]. Fortunately, these SDNCTLs can be offloaded from the switch, processed, and generated at a different entity. For example, the packet can be sent to the offload server which subsequently generates the "packet-in" SDNCTL. The key challenges for delegating and offloading this functionality include coordinating information exchange between the switch and the offload server in a quick, scalable, and lightweight manner.

2.3 Novel Switch Virtualization Contract

Next, we present a novel switch virtualization contract between the tenant and cloud provider. We envision that "power"-tenants, e.g. testing teams and developers, will request these "infrastructure-level" virtual switches in conjunction with bare-metal servers or to leverage more programmability within the data plane.

We propose to provide each tenant with the following abstraction of a virtual switch: a virtual switch is a device with N virtual ports (potentially connected to other devices) of predefined speeds; a virtual CPU and virtual memory with predefined speeds and capacity; and an ASIC that forwards packets (traffic) at a predefined backplane speed and that services SDNCTLs-ops with a given frequency (e.g. 10 "packet-in" SDNCTLs-operations per second). In this paradigm, tenants are able to run custom agents on these virtual switches and these custom agents interact with the tenant's controller using a network hypervisor (e.g. OpenVirtex)

3 USE-CASES AND DESIGN GOALS

Today, network virtualization is focused on leveraging virtual switches at the edge [10, 20] or leveraging a controller hypervisor [2, 15] to virtualize and share the network. Often providing high-layer abstractions that obscure the network. While these approaches to virtualizing a network enable tenants to easily provision and manage virtual infrastructure, it stifles the ability of network providers from leveraging a range of testing techniques [3, 11] and emerging functionalities [4, 17, 23]. In this section, we start by describing the different use-cases that motivate a need for lower level hardware virtualization of network devices.

3.1 Use-cases

- Switch Agent Testing [3, 6, 11, 12]: A popular software engineering technique is to have multiple teams develop identical versions of the same switch agent. The idea is that most teams will implement the functionality correctly. Thus if all versions are run on the same input, then the most popular output is the correct output. A switch-virtualization platform can be used to provide each version with the abstraction of sole occupancy while simultaneously distributing events to the different versions and comparing the output from the different versions.
  Similarly, the same principles can be used to enable switch agent A/B Testing [3]: to validate a switch agent before an update.
- Headless Switch Agent Updates [18]: Upgrades to the switch agent codebase must be followed by a switch agent reboot. Currently, due to the lack of separation, such events eliminate switch agent state and requires the switch agent to recreate this state. To enable, headless switch agent updates – we need to both the new and the old switch agent simultaneously and prime the state of the new switch agent before making the switch. A switch-virtualization platform is a fundamental requirement for such a system.

4 ARCHITECTURE

Switch-Visor virtualizes SDN switches enabling different tenants to run local agents on the switch, in an isolated fashion, while providing performance guarantees over interactions between these agents and the underlying switch data plane (ASIC). Switch-Visor provides each tenant with the abstraction of a network of bare metal physical switches with predefined CPU, Memory, and ASIC resources and enables the tenant to install a custom switch agent.

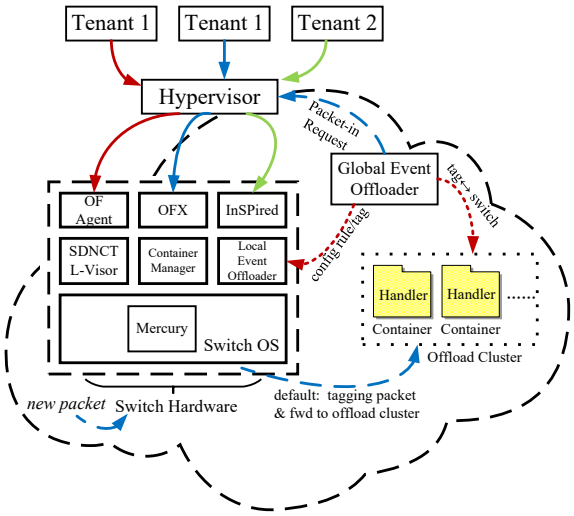


Figure 2: System Architecture

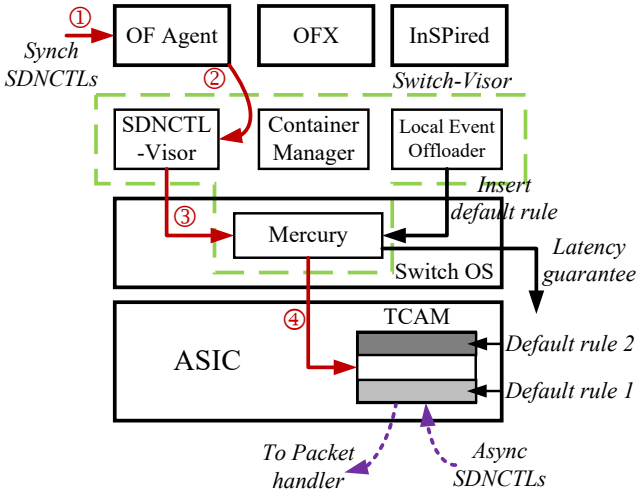


Figure 3: Switch Architecture

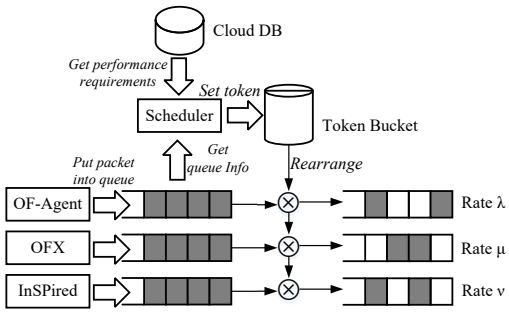


Figure 4: Rate Limiter

To achieve this functionality, Switch-Visor (Figure 2) includes four components. A traditional *container manager*, e.g., Docker, for virtualizing CPU/Memory resources and hosting a tenant's switching agents. The *SDNCTL-Visor* which virtualizes the ASIC isolating different tenants with respect to the ASIC and providing guarantees on the different SDNCTL performed on the ASIC. The SDNCTL-Visor also controls scheduling of the different SDNCTLs and rate-limits these SDNCTLs to ensure that guarantees are met. A sister component to the SDNCTL-Visor is the Mercury [5] component which focuses on TCAM and introduces TCAM management techniques to ensure performance guarantees. The final component, Event-Offloader, enables the Switch-Visor to make guarantees by eliminating uncontrollable ASIC-generated interrupts and offloading these interrupts to cheaper x86 devices with more powerful CPUs.

4.1 SDNCTL-Visor

Our main component for virtualizing a switch's ASIC, the SDNCTL-Visor, intercepts all SDNCTL messages (red arrows in Figure 3) and employs a combination of rate limiting and event scheduling to ensure that performance guarantees are maintained. In our description of SDNCTL-Visor, we assume that an admission control system is used to ensure that Switch-Visor does not support more tenants than the switch can handle.

At a high level, SDNCTL-Visor (depicted in Figure 4) intercepts SDNCTL from the different agents and distributes them into different per-tenant queues. Each queue is associated with a leaky-token bucket that enables SDNCTL-Visor to control the rate at which SDNCTL from each queue is processed.

To provide fine-control over scheduling and rate limiting of SDNCTLs, we leverage existing black box models of SDNCTL resource requirements from Tango [13]. This model generates, for diverse switches, a mapping of resource consumption for each specified SDNCTL. SDNCTL-Visor uses this model to perform admission control.

For scheduling the SDNCTLs, SDNCTL-Visor builds on the extensive literature on CPU scheduling, such as (weighted) max-min fairness with different accuracy (e.g., round-robin) [19], weighted fair queuing [8]. In our design, different SDNCTLs may have different dominant resources, such as *flow-mod* requires TCAM resources but *get-status* requires CPUs. To achieve multiple resource allocation, we use Dominant Resource Fairness (DRF) [9] which is a generalization of the max-min fairness for multiple resources to calculate the actual rate limits. Moreover, the resource requirements of different SDNCTLs may vary over time, in view of this, the algorithm must periodically re-evaluate requirements and accordingly perform resource allocations. Recall, resources utilization depends on the current state of the hardware and the set of contenting action. Fortunately, existing switch benchmarking and profiling tools [13] allow us to determine the requirements for each SDNCTL under different conditions.

4.2 Event-Offloader

While the SDNCTL-Visor controls the tenant-initiated SDNCTL, it can't control the hardware generated SDNCTL and thus the hardware generated SDNCTL can impact Switch-Visor ability to make performance guarantees. For this class of SDNCTL, we argue

that rather than dealing with them at the switch that we offload them to a cluster of cheaper and more powerful x86 servers. These offload servers can process the SDNCTL or send them to the agents on the switch or the hypervisor via TCP connections. The offload servers provide Switch-Visor with a fulcrum of control. With these servers, Switch-Visor is able to: (1) rate limit interactions from hardware to the switch and (2) provide tenants with an elastic solution to create hardware generated events. [2]

Unlike mobile offloading, offloading hardware generated SDNCTL requires a novel offloading method that avoids using local switch CPU (or memory) resources because using these resource would generate the exact problem we are trying to avoid. This novel switch-specific offloading method needs to address several challenges: (1) coordinating offload between the switch and the offload server and exchanging appropriate metadata (2) ensure that offloading occurs in an efficient and effective manner.

To address these challenges, we explore a domain-specific design. Specifically, we add tags into the packets to transfer information between the switch and the x86 server: these tags can carry sufficient data to create the event. For example for "Packet-In", the tags will carry switch-ID and in-port We also modify the SDN rules such that the hardware generated packets are not sent to the switch's CPU but rather forwards them along pre-specific paths to the x86 server. Our Event-Offloader modules provide this functionality with three components:

Global-Event-Offloader: The first, the global Event-Offloader (red solid arrow in Figure 2), that coordinates between the local Event-Offloader running on the switches and the offload servers in the x86 cluster. The global Event-Offloader assigns each switch with a specific (and unique) tag and distributes a map of $< tag, switch >$ to the servers in the clusters.

The global Event-Offloader also calculates and install each path from each switch to the offload cluster, these paths are specially created to only carry offload traffic to the clusters. Next, we discuss these paths in more detail.

Local-Event-Offloader: Given the tags and information about a set of default paths to the offload cluster, the Local-Event-Offloader creates and installs two forwarding rules into its flow table. The first rule is the lowest priority rule, and it matches all packets that match no other rule, then adds the preassigned tag to these packets and sends them along the path to the offload cluster. This rule is used to offload a packet and to add information required by the offload server to appropriately perform its offload functionality. The second rule matches any packet with any offload Tag, and forward this along to the cluster. This rule is used to continue the forwarding of offload packets from another switch. To ensure that offloading is efficient, the switches use ECMP to balance offload traffic across the different predefined offload paths.

Offload Cluster: The offload cluster consists of a set of x86 machines running containers. Each container runs code to process the offload packets. For example, upon receiving a tagged rouge packet, the offload cluster would serialize the packet into a "packet-in" event, use the tag to determine the source switch and in-port, and sends this event off to the to the network hypervisor (shown by the blue dotted arrow in Figure 2) or the tenant's controller. Additionally, the cluster can scale up or down the number of containers to handle drastic changes in the number of events being offloaded.

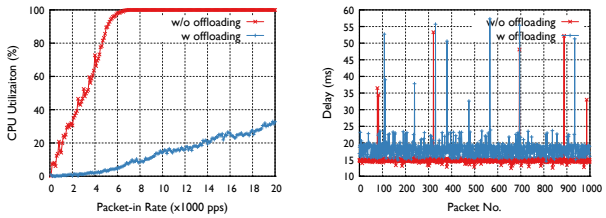
[2]Hardware generated events are often turned off because of their impact on the switch CPU, Switch-Visor eliminates this concern.

5 PRELIMINARY ANALYSIS

Offloading devices from the switch to an x86 server introduces two main overheads: first, additional latency which prior work [20] has shown to be negligible because the faster CPU at the x86 servers makes up for the additional network latency, and second, the Event-Offloader requires a fleet of x86 servers to host the containers – we note that the size of the fleet is proportional to the amount a client is willing to pay. We implemented Switch-Visor in python and deployed it on a small testbed to understand the overheads introduced by Event-Offloader.

Methodology We tested the Event-Offloader in a rack of 36 servers connected by a switch with 1 Gbps ethernet links and each server is running a virtual switch. To evaluate the performance of Event-Offloader, we conducted two experiments, firstly we initialized a number of *packet-in* events per second from $src1$ to dst with and without the running of Event-Offloader, to see what is the CPU utilization of the virtual switch, the number of *packet-in* is varied. The second experiment is that we measured the first packet delay of *ping* command from $src1$ to dst, to see how Event-Offloader will affect the packet delay.

Preliminary Results Figure 5 shows the benefit and overhead caused by offloading the hardware generated SDNCTLs.



(a) CPU Utilization (b) Packet Delay

Figure 5: Impact of Offloading

Figure 5(a) demonstrates that Event-Offloader significantly reduces the CPU utilization of the switch. Moreover, even at high packet rates, switch CPU is kept minimal. Finally, Event-Offloader allows the switch to support a higher rate than it traditional would support.

From Figure 5(b), we observe Event-Offloader introduces 30% latency (approximately 5ms) and we attribute this latency to our current implementation in python. We anticipate significant reductions when we rewrite our Event-Offloader in C++.

6 RELATED WORKS

Network Virtualization Existing approaches to virtualizing SDN switches focus on partitioning flow tables entries and enforcing strict controls overflow spaces (the type of flow tables entries that tenants can insert) [2, 15]. In this work, we take a more holistic approach and argue that in addition, to flow table isolation, switch

virtualization should provide performance guarantees across CPU, Memory, and ASIC. This enables us to truly provide each tenant with the abstraction of sole ownership over a predictable switch. Today, compute virtualization broadly falls into one of two campuses: Virtual Machine-based and Container-Based virtualization. We explore the use of containers to virtualize CPU/Memory and introduce novel primitives and abstractions to provide control over ASIC resources

CPU Offloading Several works [10, 20] include approaches to offload asynchronous messages. Scotch [20] moves the packet-in processing logic from hardware switches to an overlay of virtual switches to avoid packet-in messages from overloading the switch's CPU. Orthogonally, Mazu [10] introduces a proxy to handle packet-in and packet-out messages. While Scotch and Mazu leverage static resources for offloading, Switch-Visor uses cloud resources which enables Switch-Visor to elastically adjust to load.

Switch Benchmarking [7, 10, 13] measure, analyze, and develop black box models of switch resource utilization consumption. Our work builds on the black box models presented in these work [10, 13].

7 CONCLUSION

Today, network virtualization fails to provide infrastructure-level virtualization of SDN switches instead network virtualization provides guarantees over the traffic. In this work, we explore an alternate design space with the controversial argument that the community should push towards lower level network virtualization over control-level virtualization. A level of virtualization that requires tenants to manage their virtual switches in a manner similar to IaaS — in essence, infrastructure-level switch virtualization. Specifically, we argue for virtualizing the switch CPU and Memory using traditional methods (e.g. containers) and present a set of methods for virtualizing the switch's ASIC and interactions between the CPU and ASIC. Our system, Switch-Visor, is the first step towards Infrastructure-level Switching as a Service (ISaaS) virtualization. Our approach is realizable without any hardware changes making it immediately applicable to the current generation of deployed switches.

ACKNOWLEDGMENTS

We thank the anonymous CAN reviewers for their invaluable comments. This work is partially supported by NSF grant CNS-1409426, National Basic Research Program of China 2013CB329103, NSFC fund 61671130, 61271165.

REFERENCES

[1] [n. d.]. OpenFlow Switch Specification v1.0.0. http://archive.openflow.org/documents/openflow-spec-v1.0.0.pdf. ([n. d.]).
[2] Ali Al-Shabibi, Marc De Leenheer, Matteo Gerola, Ayaka Koshibe, Guru Parulkar, Elio Salvadori, and Bill Snow. [n. d.]. OpenVirteX: Make your virtual SDNs programmable. In *Proceedings of ACM HotSDN 2014*.
[3] Richard Alimi, Ye Wang, and Y. Richard Yang. [n. d.]. Shadow Configuration As a Network Management Primitive. In *Proceedings of ACM SIGCOMM 2008*.
[4] Roberto Bifulco, Julien Boite, Mathieu Bouet, and Fabian Schneider. [n. d.]. Improving sdn with inspired switches. In *Proceedings of ACM SOSR 2016*.
[5] Huan Chen and Theophilus Benson. [n. d.]. The Case for Making Tight Control Plane Latency Guarantees in SDN Switches. In *Proceedings of ACM SOSR 2017*.
[6] Liming Chen and Algirdas Avizienis. 1995. N-version programminc: A fault-tolerance approach to rellablllty of software operatlon. In *Fault-Tolerant Computing, 1995, Highlights from Twenty-Five Years., Twenty-Fifth International Symposium on*. IEEE, 113.
[7] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. [n. d.]. DevoFlow: Scaling Flow Management for High-performance Networks. In *Proceedings of ACM SIGCOMM 2011*.
[8] Alan Demers, Srinivasan Keshav, and Scott Shenker. [n. d.]. Analysis and simulation of a fair queueing algorithm. In *Proceedings of ACM SIGCOMM 1989*.
[9] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. [n. d.]. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *Proceedings of USENIX NSDI 2011*.
[10] Keqiang He, Junaid Khalid, Sourav Das, Aditya Akella, Erran Li Li, and Marina Thottan. [n. d.]. Mazu: Taming latency in software defined networks. *University of Wisconsin-Madison Technical Report, 2014* ([n. d.]).
[11] Eric Keller, Minlan Yu, Matthew Caesar, and Jennifer Rexford. [n. d.]. Virtually Eliminating Router Bugs. In *Proceedings of ACM CoNEXT 2009*.
[12] P. Khanduri. [n. d.]. Diffy: Testing services without writing tests. ([n. d.]).
[13] Aggelos Lazaris, Daniel Tahara, Xin Huang, Erran Li, Andreas Voellmy, Y Richard Yang, and Minlan Yu. [n. d.]. Tango: Simplifying SDN control with automatic switch property inference, abstraction, and optimization. In *Proceedings of ACM CoNEXT 2014*.
[14] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. [n. d.]. Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency. In *Proceedings of ACM SOCC 2014*.
[15] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. [n. d.]. Can the Production Network Be the Testbed?. In *Proceedings of USENIX OSDI 2010*.
[16] Christian Sieber, Andreas Blenk, Arsany Basta, and Wolfgang Kellerer. [n. d.]. hvbench: An open and scalable SDN network hypervisor benchmark. In *Proceedings of IEEE NetSoft 2016*.
[17] John Sonchack, Adam J Aviv, Eric Keller, and Jonathan M Smith. [n. d.]. Enabling practical software-defined networking security applications with ofx. In *Proceedings of NDSS 2016*.
[18] Laurent Vanbever, Joshua Reich, Theophilus Benson, Nate Foster, and Jennifer Rexford. [n. d.]. HotSwap: Correct and Efficient Controller Upgrades for Software-defined Networks. In *Proceedings of the ACM HotSDN 2013*.
[19] Carl A Waldspurger and William E Weihl. [n. d.]. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of USENIX OSDI 1994*.
[20] An Wang, Yang Guo, Fang Hao, TV Lakshman, and Songqing Chen. [n. d.]. Scotch: Elastically scaling up SDN control-plane using vswitch based overlay. In *Proceedings of ACM CoNEXT 2014*.
[21] Guohui Wang and T. S. Eugene Ng. [n. d.]. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of IEEE INFOCOM 2010*.
[22] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. [n. d.]. Bobtail: Avoiding Long Tails in the Cloud. In *Proceedings of USENIX NSDI 2013*.
[23] Ji Yang, Zhenyu Zhou, Theophilus Benson, Xiaowei Yang, Xin Wu, and Chengchen Hu. [n. d.]. FOCUS: Function Offloading from a Controller to Utilize Switch Power. In *Proceedings of IEEE NFV-SDN 2016*.