InFEP – Lightweight Virtualization of Distributed Control on White-box Networking Hardware

Thomas Kohler, Frank Dürr, Christian Bäumlisberger, and Kurt Rothermel
Institute of Parallel and Distributed Systems, University of Stuttgart, Germany
{firstname.lastname}@ipvs.uni-stuttgart.de

***Abstract*—Recent developments in networking hardware and software-defined networking have enabled full distribution of network control to reduce control latency and increase reliability. However, both, hardware and software of current white-box networking hardware are highly heterogeneous, which limits the deployment and operation of switch-local control applications. Furthermore, switch-local control raises yet unconsidered security concerns.**

In this paper, we present our concept of in-forward-element processing, which leverages the open access to the control plane of white-box networking hardware to deploy control logic directly onto switches. We combine local control applications with lightweight virtualization to cope with networking hardware heterogeneity and to achieve required isolation properties and ease of management. Beyond distributed network control, we show this scheme is also beneficial for implementing switch-local virtual network functions (NFV), processing packets. Highlighting the practicability of the concepts, we provide an overview of the current white-box networking hardware and software landscape and their compatibility with lightweight virtualization technologies. To this end, we perform an empirical evaluation of NOS-virtualization combinations on such hardware and compare the results with respect to incurring virtualization overhead.

I. Introduction

Alongside the proliferation of Software-defined Networking (SDN), recent years have seen an increasing trend towards white-box networking. In traditional black-box switches, the control plane is tightly coupled to the underlying hardware and is only accessible through proprietary CLIs or APIs and, in case of SDN support, an interface for remote programmability of the data plane behavior, e.g., through the popular OpenFlow protocol. SDN separates the data plane from the control plane, allowing for control decisions based on a global network view typically taken remotely in the control plane. Similarly, white-box networking decouples the data plane, where specialized hardware (typically ASICs) process packets, from the control plane, where control-software determines the behavior of this processing. In white-box networking, the switch hardware is independent and is not tied to the so-called network operating system (NOS). Typical white-box networking NOSes consist of a standard Linux OS and control software running atop, forming the switch's control plane. Similar to the transition in the server market from proprietary server software and hardware to open operating systems like Linux on commodity off-the shelf hardware (COTS), or the decoupling from proprietary hardware (softwarization) of network functions (NFs) in Network Function Virtualization (NFV), white-box networking offers superior flexibility at greatly reduced capital expenditures. While white-box switches typically feature the same data plane processing hardware (switch silicon) as proprietary products, their computing resources on the control plane have reached a level comparable to small workstations and are still becoming increasingly powerful. Contrasting black-box switches, white-box NOSes are completely accessible, allowing for the execution of arbitrary applications on their control plane.

These properties allow for the exploitation of the switch's locality. SDN is based on the paradigm of *logically* centralized control of network elements. Well-known from the domain of Distributed Systems, logical centralization implies hiding the complexity of a physically distributed system from the application (*distribution transparency*). In particular, network control logic in SDN has a global view on the network and is implemented by possibly distributed control applications. In most SDN architectures, any control decision is taken at a remote SDN controller. For some control decisions, global knowledge is however not needed, making the involvement of a centralized remote controller superfluous. Instead, such decisions can be taken locally at the switch, which we denote as *in-forwarding-element processing* (InFEP). InFEP greatly reduces control latency and thus greatly benefits processing of time-sensitive traffic, such as real-time or signaling traffic, and time-critical control mechanisms, such as link-failure recovery. In previous work [1], we have proposed an architecture for the centrally coordinated distribution of control whose implementation is publicly available at `zerosdn.github.io/`. Furthermore, we have presented control applications that can run directly on switches, taking decisions locally, while still profiting from a global view.

While appealing due to open access to a powerful control plane, employing InFEP on white-box networking hardware also entails challenges. In particular, the current white-box switch landscape exhibits a high heterogeneity with respect to hardware and software. A first differentiation is found in the switch-silicon, which differs in type, e.g. ASIC or NPU, and model. Considering forwarding performance, flexibility, capability, and accessibility, the selection of a switch silicon is crucial. A second differentiation lays in the hardware architecture of the control plane as one of x86, PowerPC, or ARM. On the software side, although all currently available NOSes are Linux-based, they differ in the aspects openness (closed- or open-source), used kernel, and Linux-distribution. Furthermore, the execution of local control logic on a switch raises concerns regarding security and reliability. Adverse behavior of local logic on a switch's control plane poses a severe threat to its entire operation. For instance, excessive resource consumption of one control application could starve essential control plane processes, such as the OpenFlow agent, which is the sole interface to the underlying switch silicon in OF-switches. In such a case, the control plane would be unable to detect and thus properly react to data plane events like port state changes, e.g., in case of link failures. Also, the network's administrative domain might differ from the origin of the control application code. For instance in NFV, the network operator typically differs from the vendor of a virtualized network function (VNF), requiring trust in the code issuer and functional correctness of the VNF. In case of network virtualization, where tenants are provided logical partitions of network resources, the origin and behavior of control logic might not even be known to the network operator. Consequently, fine-grained control of resources (CPU, RAM, storage) as well as hard isolation properties have to be implemented. Virtualization technologies perfectly meet these requirements. Beside resource control and isolation, management is greatly simplified through orchestration frameworks. However, with high virtualization overhead, the benefits of local processing, in particular the greatly reduced latency, would be put at stake.

This paper presents a concept for network processing directly on the switches supporting both, distributed control of SDN networks including switch-local control function and on-switch processing of virtual network functions. We leverage the open access to the control plane of white-box networking hardware for deployment and operation of control logic. We combine local control applications with lightweight virtualization to cope with white-box networking heterogeneity and to achieve required isolation properties. Essential for our concept of InFEP, we elaborate on our position of defining the sweet spot where packet processing in a network should happen. In light of this question, we depict the points of view of SDN, middleboxing, and NFV, relate these paradigms, and show benefits of mutual adoption. Highlighting the practicability of InFEP, we provide an overview of the current white-box networking hardware and software landscape and the compatibility with lightweight virtualization technologies. To this end, we performed an empirical evaluation of NOS-virtualization combinations on a white-box switch and compare the results with respect to the introduced virtualization overhead.

The remainder of this paper is structured as follows: We describe the system model in §II and proceed with the discussion about placement of packet processing and the relationship of InFEP, middleboxing and NFV in §III. We introduce our concept of InFEP in §IV and concretize requirements on lightweight virtualization. In §V we give an overview over the white-box landscape, and present conducted evaluations in §VI. We show related work in §VII before we conclude and give an outlook on future work in §VIII.

II. System Model – Distributed SDN Controller

Our system model assumes an event-based distributed SDN controller architecture as introduced in our previous paper [1] and depicted in Figure 1. End-hosts (h_i, undepicted) and forwarding elements (FE), interchangeably called switches (S_i), constitute the data plane (DP). The control plane (CP) is responsible for taking control decisions whose results are pushed down to the data plane in form of forwarding rules, to implement the desired network behavior. The control plane can also be used to directly process data plane packets. Network control logic is split into lightweight control modules, named *controllets* (CM_i), running in dedicated processes, possibly on separate control plane hardware such as COTS servers (H_i, undepicted). The lightweight nature of controllets also allows

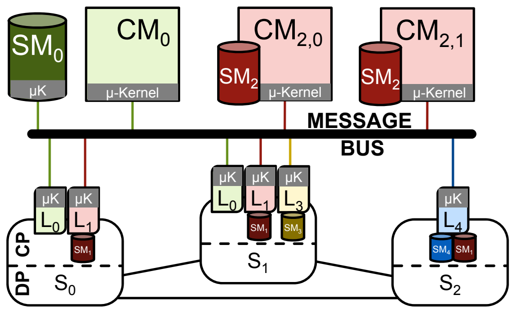

Figure 1. System model of an event-based distributed SDN controller architecture with switch-local (L_i) & external controllets (CM_i), interconnected by a message bus constituting the control plane and interconnected switches (S_i) in the data plane.

for an execution directly on switches. We denote switch-local controllets as L_i. Overall, *CM* and L form the control plane. In this paper, we concentrate on switch-local controllets.

Controllets consist of two components: a so-called micro-kernel (μK) that just provides basic functions for interaction with other controllets, and the actual control logic, e.g., network topology management or routing. They communicate through messages over a unified *message bus*, spanning the control network. The message bus offers rich communication paradigms such as publish/subscribe, broadcast, or direct peer-to-peer communication. Furthermore, the message bus is responsible for guarantees on message delivery, such as "exactly once" or "at least once". Controllets subscribe to events in the data plane, such as packet ingress or state changes of ports, and to control events, such as notifications of joining or leaving controllets or changes in the global view of the network.

III. Background & Discussion

Many network functions (NF), such as load-balancing or firewalling, depend on exerting fine-grained control over network traffic and ultimately boil down to providing connectivity—or deliberately not providing connectivity—and thus to forwarding behavior which nowadays SDN is able to flexibly control. Although enabling powerful NFs, the middleboxing model, which also *network function virtualization* (NFV) follows, comes with inherent disadvantages, which we describe in the following by discussing the questions: *Where should packet processing ideally happen and how powerful is in-forwarding-element processing these days?*

To discuss the placement of packet processing, we consider two criteria on an entity that implements a NF by performing packet processing (*packet processor*): 1) expressiveness and 2) performance, i.e., processing latency and throughput. Latency is directly affected by processing power and by the *processing distance* (d), which we informally define as the number of hops between the traffic (source, sink and path) and the packet processor affecting this traffic. We differentiate between *in-situ* processing, where packets are processed by network elements (NE) that are within their unmodified, typically shortest path between source and sink, and *ex-situ* processing, where traffic has to be artificially steered through remote packet processors. We consider the following placements, sorted by processing distance ascendingly.

1) End-hosts have been recently proposed to process packets directly at their source/sink [2]. While this *in-situ* approach in fact has the shortest distance ($d = 0$), it logically extends the control plane to end-hosts thus adding another level of complexity to network management. End-hosts should prioritize hosting applications and providing them *access* to network services over implementing these network services. Thus, access and performance should be isolated. Two implementations of this model, commonly referred to as SmartNICs, are Microsoft's FPGA-based pre-NIC processors[1] and programmable NICs, such as the popular Netronome platform. However, hardware-near programming is tedious and typically lacks generalizability.

2) FEs (switches) are the first type of NEs traffic encounters on its path from source to sink and thus have the shortest processing distance ($d = 1$) of end-host external placements. In the FE data plane, packets are processed on highly optimized hardware providing line-rate processing with low latency and jitter. Due to *in-situ* processing, no additional communication latency occurs. We thus argue, in-line with the proactive SDN control scheme, that FEs are the optimal place for packet processing. We present our approach InFEP in detail in §IV-A.

A traditional, non-SDN example in traffic engineering is *Equal-cost Multipath Routing* (ECMP), where link redundancy is exploited by switch-local forwarding[2] without external control enabled by integrating hashing capabilities in switch silicon. The expressiveness of FE's data plane processing depends on its programmability, which is limited though, such that not all NFs can be implemented directly in the FE data plane. OpenFlow-enabled switches employ match-action processing semantic, where matching enables fine-grained classification of traffic (flows) on which actions including modification of packet headers and switch-port output are applied. This semantic is however powerful enough to cover implementation of stateless firewalls, load balancing, monitoring, etc. Even complex appliances, such as content-based routing in the domain of communication middleware can be entirely substituted by employing match-action processing within FEs, providing line-rate throughput [3] and eliminating the need for a remote middlebox ("broker"). Recent advances in the field of data plane programmability like programmable network processors (NPUs) or FPGA-based switches in combination with software frameworks exploiting the advanced processing capabilities, like the popular P4 [4] initiative, further push this frontier. Despite its potential of being a disruptive technology, it has not yet been widely adopted, though.

3) Remote hardware is used for NF implementation in both **middleboxing** (proprietary appliances on closed hardware) and **NFV** (general-purpose hardware running virtualized software NF-implementations). On the one hand, packet processing in software running on general-purpose hardware has become remarkably fast. Furthermore, the NFV-infrastructure (NFVI) provides powerful abstractions allowing for unified management and orchestration of virtualized network functions (VNFs). On the other hand, *ex-situ* packet processing inherently requires to re-steer the traffic to traverse additional hardware entities—hence the name *middle*-boxes. While dedicated middleboxes are placed on or at least close to the path, virtualization hosts providing VNFs are typically remote, i.e., off the path. Furthermore, in modern networking, traffic typically has to traverse multiple (V)NFs (*service chaining*). Thus, $d > 1 + n$, where n is the chain length. Although NFV may mitigate chaining costs through consolidation of multiple VNFs into a single physical host, the traffic still has to traverse multiple software components (virtual switches, hypervisors, virtual NICs). Overall, significant latencies accrue depending on physical or logical distance of (V)NFs and chain lengths. For WAN scenarios, incurred latency can easily reach an order of tens to hundreds of milliseconds. Furthermore, the probability of failures increases with increasing n.

[1]http://conferences.sigcomm.org/sigcomm/2015/pdf/papers/keynote.pdf

[2]Incorporating local SDN control, static ECMP can be made much more dynamic, adapting to changing link-utilization [1].

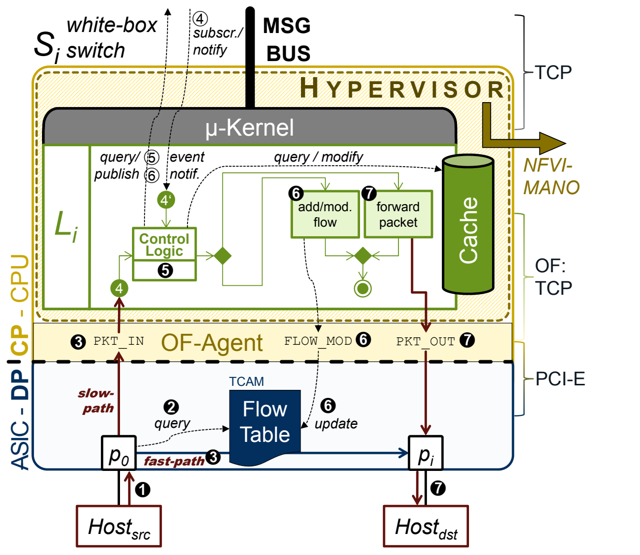

Figure 2. Hardware and software architecture of a typical ASIC-based OpenFlow-enabled white-box switch (S_i), depicting exemplary control flow and data flow and the integration of a virtualized local controllet (L_i) in InFEP.

Enabled by SDN and white-box-networking, middleboxes or VNFs can be partially replaced or augmented by increasingly powerful InFEP—in particular network-centric NFs (cf. [5]).

IV. In-Forwarding-Element Processing (InFEP)

In this section, we introduce our concept of in-forwarding-element processing (InFEP), its lightweight virtualization, and discuss mutual benefits with NFV.

A. FE-local Control Logic & Packet Processing

InFEP's meaning of "in" is twofold. Regarding the placement of packet processing (a), we argue to process traffic in SDN-enabled FEs, rather than in dedicated (virtualized) middleboxes, if possible. Regarding the role of FEs in the SDN paradigm (b), we argue to employ controllets directly on them, effectively using the FE's control plane as such, i.e., for local control decision making, rather than as a mere interface to a (centralized) remote controller, thus minimizing d, i.e., pushing controllets closer to the traffic they are affecting. Both aspects are illustrated in Figure 2 which shows a virtualized local controllet L_i on a typical ASIC-based, OpenFlow-enabled white-box switch S_i.

Aspect (a) implies that traffic is processed mainly in S_i's data plane (blue bottom-part) by the high-performance special-purpose hardware processor (here: ASIC), based on rules defined by the control plane ❻. Hence, this is called "fast-path" ❶→❷→❸→❼. Ideally, all InFEP happens here.

The control logic ❺ of a local controllet L_i that defines the rules for (a) and is the main aspect of (b) runs in S_i's control plane (yellow top-part) and is isolated by a virtualization hypervisor (c.f. §IV-B). It is invoked ❹ either by data plane events (e.g., ❸ `PACKET_IN` due to lack of a matching rule ❷) or control plane events L_i has subscribed for ④ (e.g., topology changes or joining/leaving controllets). Control logic[3] possibly involves consultation of remote controllets ⑤ (at neighbor switches L_j or CM_k) or querying/modification of the local cache, which stores data of local scope as well as cached/aggregated data of more global scope that is relevant for local decision making (e.g., topology information or global policies such as ACLs). Possible outcomes are rule changes ❻, accompanied by packet egress ❼ if triggered by a `PACKET_IN` event, the delegation of the triggering event for remote processing ⑥, or firing of a control plane event ⑥ (e.g., a change in topology due to a local link failure event).

Reconsidering the question of how powerful InFEP is, our architecture in principle allows for the implementation of arbitrarily complex NF by sending every packet to virtualized FE-local controllets in the control plane. This would however involve the traversal of two interfaces each adding latency: ASIC$\leftrightarrow$CPU, typically interconnected by a PCI-Express bus using DMA [6] and OF-Agent$\leftrightarrow$ L_i using OpenFlow over TCP over a local loop-back network interface. Although the PCI-E interface would provide sufficiently high throughput and low latency, for protection of the control plane, which is also a main motivation for using virtualization in InFEP in the first place, switch vendors intentionally limit the packet throughput (rate control) and latency on this interface. Hence, this is called the "slow-path". Furthermore, throughput of control plane packet processing is limited due to limited, yet increasing computing resources on a white-box switch's control plane. InFEPs rationale is on the one hand to mitigate the usage of control plane resources by pushing as much processing as possible to the fast-path, but on the other hand to protect the control plane (cf. §IV-B) when the processing of packets or events cannot happen in the data plane. In WAN scenarios with high ds, i.e., high propagation latencies, this control scheme still pays off for rather low-volume but time-sensitive traffic. For data center networking we argue to restrict this scheme to events of rather low frequency that invoke control decisions rather than employing massive packet processing in the control plane. Connection setup for long-lived flows (elephant flows) and handling of link-failures or link-utilization change events in fast-failover and adaptive link load balancing, respectively, are major citizens for InFEP and described in [1].

B. Lightweight Virtualization

In our scheme, control logic is not as well-known and thus not as predictable as in the black-box switch model. Furthermore, increasing switch control plane power allows for more compute-intensive or parallel controllets, which even if triggered by rate-limited traffic may consume many resources, possibly starving other processes. Thus, control over local control logic has to be considered a crucial security aspect. Control here consists of two aspects: a) implementing **isolation** properties, to ensure data integrity by preventing access among controllets, and b) enacting prioritization and fine-grained **resource control**, to protect operation of individual controllets and other essential control plane processes (Quality of Service).

In traditional *virtualization*, the emulation of resources provides isolation while their allocation to a VM depicts resource control. Two ways to counter virtualization costs (image size, memory footprint, and boot time) have been evolving: 1) stripping down the guest OS to a bare minimum, i.e., providing just the functionality the virtualized application needs for its operation (*library OS / Unikernel*) and 2) abandoning hardware emulation and full OS virtualization in favor of using isolation features of a shared kernel, providing multiple isolated user-space instances (*Container*). NVF traditionally relies on VMs (full virtualization or unikernels) but has started looking into containers recently [7].

Unikernels thus naturally lend themselves to cloud computing and NFV where they have been gaining importance in the recent years, as for instance with ClickOS [8], a "minimalistic, virtualized operating system for network processing". They have a single address-space. Kernel and application are a single, unified process. This eliminates the need for context-switches, but also prevents usage of multi-processing, signals, dynamic libraries, and virtual memory. Furthermore, application logic has to be ported to a particular Unikernel framework. However, Rump kernel [9] uses NetBSD's kernel and libc. Thus, POSIX-compliant applications obeying these restrictions are supported without modifications. We show the performance of a state-of-the-art SDN controller running as a rump kernel in Section VI.

In our scheme, the main goal of virtualization is to protect the control plane from unintentional adverse behavior of controllets which justifies to lessen isolation. Containers allow for fine-grained control over the scope of isolation with almost no additional overhead. They rely on *namespaces*, a feature of the Linux kernel that isolates system resources, e.g., user and process IDs, IPC, filesystems and networking, of a set of processes whose resources are accounted using the control groups (*cgroups*) kernel feature. Two well-known implementations of containers are LXC and Docker. Beside the kernel, which is necessarily shared among all containers, a container configuration can share or isolate any combination of namespaces. For instance, in the evaluations we show the impact on performance of employing network namespace isolation.

C. Symbiosis of SDN & NFV

We have argued that InFEP can augment NFV. Reciprocitly, we now briefly sketch how InFEP could benefit from an integration into an NFVI, leaving the concrete implementation for future work. Just like VNFs, InFEP containers or unikernels have to be deployed (but onto switches) and being scaled as needed. The NFVI offers efficient and holistic management of the underlying virtualization infrastructure, VNF instance management, and their orchestration. White-box switches can be integrated into the NFVI as compute-virtualization resources. Thus, also NFVI's management and orchestration (mano) functionality would be extended to switch-local network logic. On the switch, a small NFVI adapter has to implement corresponding NFVI reference points (*Nf-Vi*, *Ve-Vnfm*).

V. White-box Networking Landscape

Beside the aforementioned heterogeneity of white-box hardware w.r.t. switch silicon (ASIC, NPU, FPGA) and control plane architecture (x86, PowerPC, ARM), also the software components exhibit function-critical heterogeneity. In this section, we give a short introduction to the landscape of white-box software and describe its implications on compatibility

[3] Abstracted from here. Detailed concrete examples can be found in [1].

with OpenFlow-based local control logic, requiring open control plane access and OpenFlow support, as well as with virtualization technologies, requiring a hypervisor (Unikernel, VM) and namespaces/cgroups kernel support (container).

A typical white-box network operating system (NOS) comprises the following components: 1) A base OS, typically Debian with a Linux-kernel. 2) Components for accessing platform hardware including the switch silicon, fans, LEDs, etc. Access to the switch silicon is provided through drivers, which are built against a typically proprietary silicon SDK. The top layer of the driver offers an abstracted API for configuration of the switch silicon hardware pipeline. The most relevant APIs are Broadcom's Open Network Switch Library (OpenNSL) and OpenFlow Data Plane Abstraction (OF-DPA) as well as the generic switch abstraction interface (SAI). 3) A forwarding agent that interfaces with the driver to program the data plane. Most prominent are: Indigo OpenFlow Agent (OF-DPA), SnapRoute (OpenNSL, SAI), Facebook FBOSS (OpenNSL).

Prominent NOSes include a) Open Network Linux (ONL): open-source (part of the *Open Compute Project*), broad selection of silicon drivers and forwarding agents, b) Pica8 PicOS: proprietary, based on Open vSwitch (OVS), and c) Cumulus Linux: proprietary. All named have open control plane access, however, just ONL and PicOS feature OpenFlow forwarding agents. Although InFEP is not generally tied to a specific SDN southbound protocol, we use OpenFlow in our implementation. We thus focus on ONL and PicOS for the remainder of this paper. On ONL we are able to run LXC, Docker, and qemu with KVM-acceleration, whereas on PicOS, we are able to run only qemu-KVM (which is officially supported).

VI. Evaluation

In this section, we present the evaluation of InFEP. We want to provide an overall impression of the performance of control plane processing and control logic on a white-box switch compared to remote controllers (typical SDN) and measure the overhead imposed by virtualization technologies specifically on white-box switches. Furthermore, we compare NOSes and their underlying forwarding agents. Due to space constraints we restrict the evaluation to control plane processing (InFEP aspect (b)), while referring to evaluations of SDN data plane performance (InFEP aspect (a)), specifically to [3] for the SDN-based pub/sub middleware appliance.

The **device under test** is a typical top-of-rack white-box switch Edge-Core AS5712-54X, whose hardware specification is publicly available under the Open Compute Project. Its control plane comprises an x86 Intel Atom CPU with 4 cores

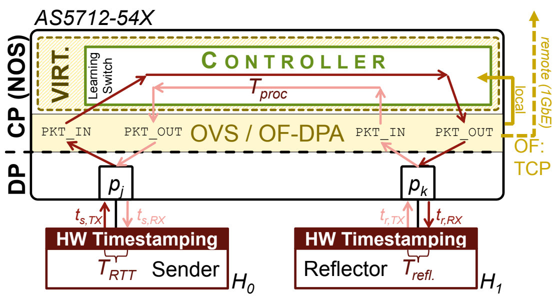

Figure 3. Evaluation setup of measuring control plane processing latency (T_{proc}) on a white-box switch using hardware-timestamping on end-hosts.

at 2.4 GHz, 8 GB RAM, and a 1 GbE NIC. Atop we run *ONL* 2.0 with a 3.16.39-LTS kernel and *PicOS* 2.8 with a 3.16.7 kernel. On the data plane, it features a Broadcom Trident II ASIC with 48×10 GbE and 6×40 GbE ports.

We evaluate a set of **SDN (OpenFlow 1.3) controllers**: NOX (*nox13oflib*), our "Autonomous Forwarding Controllet" (*ZSDN*) [1], both written in C++, and the Python-based *Ryu*. While ZSDN is a full InFEP implementation (forwarding and ACL), we use the other controllers to emulate the forwarding NF through switch-local execution of a learning switch controller module. We evaluate **virtualization overhead** of *Docker* as well as rump kernels (*rumprun*) and full VMs (*KVM*), both running on QEMU with KVM-acceleration enabled by the Atom's VT-x support. The baseline is bare-metal execution (*none*). We omit LXC since it is technically equivalent to Docker. Since NOX and ZSDN are relying on Digital Shared Objects (DSO), we were not able to port them to rumprun. Replacing DSOs would have been too intrusive.

Our **methodology**, illustrated in Figure 3, is as follows. We run a combination of NOS, virtualization and controller on the switch's control plane. For the evaluation, we provoke that every ingress packet in the data plane is processed in the control plane. To this end, the controllers run learning switches, but do not install flows. In the data plane, we connect two end-hosts (H_0, H_1) with 10GbE links to the switch. H_0 is sending packets to H_1 where they are reflected back. Packet identity is ensured through unique sequence numbers attached to the packets (as sole payload). Both, egress (t_{TX}) and ingress (t_{RX}) times are captured using hardware-timestamping. Thus we can calculate the RTT at the sender ($T_{RTT} = t_{s,RX} - t_{s,TX}$) and the time spent for reflection at the reflector ($T_{refl} = t_{r,TX} - t_{r,RX}$) with high precision. We acquire the (one-way) switch processing latency as $T_{proc} = 1/2 * (T_{RTT} - T_{refl})$. Transmission and propagation delay are negligible.

As expected, **packet throughput** is limited. ONL caps at 1kpps with a low peak CPU utilization of the OF-DPA daemon of 50% of one core. This shows that the rate limit is clearly not caused by a CPU bottleneck. PicOS behaves differently. For switch-ingress rates $\geq$ 20kpps, we measure an egress-rate of about 7kpps, while the OVS daemon consumes two cores.

For evaluating **switch control plane processing latency** (Figure 4), we send with a rate of 100pps for 50s. Through reflection, the effective packet rate (ingress rate at the switch) is doubled. We begin with a **comparison of controllers** running bare-metal on ONL or remotely (Xeon E5-1650v3, 6 cores at 3.50GHz, connected over 1GbE). Since the figures of nox13oflib are within 3% of ZSDN, we combine them henceforth. Overall, the python-based Ryu is expectedly performing worse than its C++ counterparts. With $330 \pm 75\mu s$, they are considerably faster and deviate far less than Ryu with $1795 \pm 225\mu s$. Remote execution is about 1.8 times slower with NOX/ZSDN, whereas remote Ryu execution is surprisingly about 20% faster.

We find that Docker has the lowest **virtualization overhead**. With full isolation of all but the network namespace, **Docker** imposes almost no overhead for NOX/ZSDN. Latency and its deviation, are within $1\mu s$ to bare-metal execution. For Ryu, about $80\mu s$ are added to latency. Isolating the network namespace (undepicted) incurs $110\mu s$ additional latency.

Next, we measure the combined overhead of the hypervisor and the guest OS of virtualization variants. **Full virtualization** (*KVM*) adds large overhead. On average, $410\mu s$ (factor 1.5) incur for ONL and $820\mu s$ (factor 1.8) for PicOS, both slower

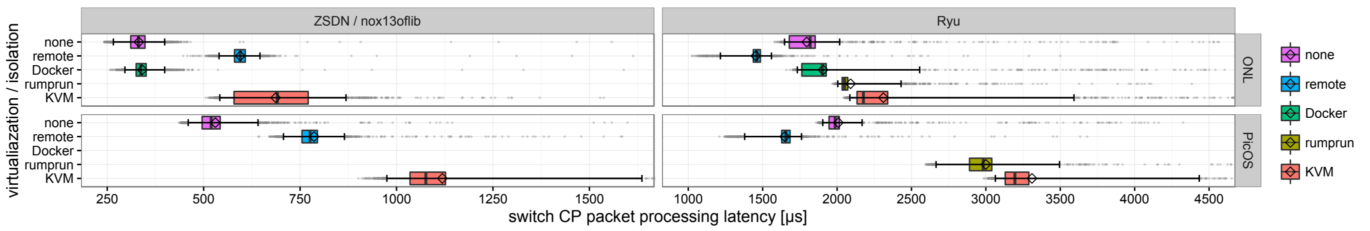

Figure 4. Processing latencies (x-axes: medians (bars), averages (diamonds)) of controllers (grid horizontal) running locally on network operating systems (grid vertical) with varying isolation mechanisms (y-axis). Bare-metal (*none*) and *remote* execution are given as baselines. Whiskers enclose 95% of measured values.

than remote execution. Standard deviations are larger by factors 2 and 2.2, respectively. As guest network device, we are using *virtio*, a pseudo-paravirtualized driver that runs within KVM. Userspace network emulation (SLiRP) adds additional $410\mu s$.

Ryu as a **Unikernel** (*rumprun*) is showing much better results. Compared to a full VM, latencies and deviations are greatly reduced by $220\mu s$ and $285\mu s$ for ONL, and $310\mu s$ and $190\mu s$ for PicOS. This is the result of the minimal guest OS and hence reduced OS overhead. Compared to bare-metal/Docker, factors of 1.2 and 1.5 for latency on ONL and PicOS are promising. We find that the standard Linux bridge in use at least partially accounts for the larger overhead. By using optimized software bridges, like macvlan, VALE or OVS in combination with SR-IOV (paravirtualization), latencies as small as $45\mu s$ [8] can be achieved—on server hardware, though.

Lastly, we evaluate the difference between the **NOSes**. For all measurements, compared to ONL, PicOS adds quite consistent latency of $200\mu s$ on average for bare-metal and remote, $420\mu s$ for NOX/ZSDN, and as high as $950\mu s$ for KVM and rumprun. Especially for higher packet rates, we have observed instability of QEMU on PicOS. One source of the discrepancy may lay in the forwarding agent. While OF-DPA (used in ONL) tightly reflects the underlying switch silicon hardware pipeline, which is quite restrictive, OVS (used in PicOS) offers an abstracted and unrestricted multi-table pipeline as per the OpenFlow 1.3 specification. Implementing this mapping surely causes additional latency. It can be assumed that this would account for a rather static offset, though.

We can conclude that containers provide isolation as needed at minimal cost. We could verify and quantify the benefit of reduced latency to be almost halved with containerized local control logic, despite isolation. Note, that our scenario of a one-hop switched 1GbE control network, is almost ideal, providing a lower bound for switch control plane processing latency. For larger distances or WAN scenarios, remote control latencies are expected to be orders of magnitudes higher, even compared to local yet sub-optimal virtualization variants.

VII. Related Work

We now describe related work other than already mentioned.

OpenBox [5], an SDN-based framework for NFs supports aspect (a) of InFEP, to employ SDN-switches as packet processors in favor of middleboxes, however not its other aspect (b), since OpenBox assumes a centralized controller.

E2 [10] proposes a framework for NFV applications, strengthening the role of SDN in NFV management through the unification of SDN and NFV in a single controller that automates NF-placement and service interconnection (management and orchestration). For extended network processing capabilities, they push richer programming abstractions into the network layer, however relying on software switches.

Along that line, P4 [4] exploits extended semantics of data plane processing, resulting in more expressive packet processing, supporting InFEPs argument of in-data-plane processing. However, control plane distribution (switch-local logic) is not addressed.

VIII. Summary & Future Work

In this paper, we have presented our concept of in-forwarding-element processing (InFEP) and have discussed its relation to middleboxing and NFV. We have implemented isolation and resource control of switch-local control through lightweight virtualization and shown its practicability and performance on white-box networking hardware. Future work includes porting an InFEP SDN controller to a Unikernel and improving latency of the switch's hypervisor network back-end.

References

[1] T. Kohler, F. Dürr, and K. Rothermel, "ZeroSDN: A Highly Flexible and Modular Architecture for Full-range Network Control Distribution," in *2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. IEEE Press, May 2017, pp. 25–37.

[2] H. Ballani, P. Costa, C. Gkantsidis, M. P. Grosvenor, T. Karagiannis, L. Koromilas, and G. O'Shea, "Enabling End-Host Network Functions," in *Proceedings of the 2015 ACM SIGCOMM Conference*, ser. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 493–507.

[3] S. Bhowmik, M. A. Tariq, B. Koldehofe, F. Dürr, T. Kohler, and K. Rothermel, "High Performance Publish/Subscribe Middleware in Software-Defined Networks," *IEEE/ACM Transactions on Networking*, vol. PP, no. 99, pp. 1–16, 2017.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-independent Packet Processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, pp. 87–95, Jul. 2014.

[5] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 511–524.

[6] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan, "Measuring Control Plane Latency in SDN-enabled Switches," in *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, ser. SOSR '15. New York, NY, USA: ACM, 2015, pp. 25:1–25:6.

[7] R. Cziva, S. Jouet, K. J. S. White, and D. P. Pezaros, "Container-based network function virtualization for software-defined networks," in *2015 IEEE Symposium on Computers and Communication (ISCC)*, Jul. 2015.

[8] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "ClickOS and the Art of Network Function Virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, Berkeley, CA, USA, 2014, pp. 459–473.

[9] A. Kantee and J. Cormack, "Rump kernels: No os? no problem!" in *;Login: USENIX Magazine, October 2014, Vol. 39, No. 5*. USENIX.

[10] S. Palkar, C. Lan, S. Han, K. Jang, A. Panda, S. Ratnasamy, L. Rizzo, and S. Shenker, "E2: A Framework for NFV Applications," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 121–136.