The Power of Composition: Abstracting a Multi-Device SDN Data Path Through a Single API

Stefan Geissler*, Stefan Herrnleben*, Robert Bauer‡, Alexej Grigorjew*, Thomas Zinner*, Michael Jarschel§
*University of Würzburg, Germany
Email: {stefan.geissler|stefan.herrnleben|alexej.grigorjew|zinner}@informatik.uni-wuerzburg.de
‡Karlsruhe Institute of Technology, Germany
Email: robert.bauer@kit.edu
§Nokia Bell Labs, Germany
Email: michael.jarschel@nokia-bell-labs.com

***Abstract*—Software Defined Networking aims to separate network control and data plane by moving the control logic from network elements into a logically-centralized controller. Using a well-defined, unified control-channel protocol, such as OpenFlow, the controller is able to configure the forwarding behavior of data plane devices. Here, the OpenFlow protocol is translated to vendor- and device-specific instructions that, for instance, manipulate the flow table entries of a switch. In practice, SDN-enabled switches often feature different hardware capabilities and configurations with respect to the number of flow tables, their implementation, and which kind of data plane features they support. This leads to device heterogeneity within the SDN landscape, thereby obstructing the increased scalability and flexibility promised by the SDN paradigm. To overcome this challenge we propose TableVisor, a transparent proxy-layer for the SDN control channel that enables the flexible abstraction of heterogeneous data plane devices into a single emulated data plane switch. In this paper, we extend our previous work by introducing features to integrate modern P4 devices into an existing SDN environment and perform a detailed performance evaluation to quantify the overhead induced by our approach.**

***Index Terms*—Software-defined networks, data plane abstraction, programmable hardware, p4**

I. Introduction

SDN-enabled switches often feature different hardware capabilities and configurations with respect to the number of flow tables, their implementation, and which kind of data plane features they support [1], [2], [3], [4], [5], [6]. From the perspective of SDN application developers, the lack of devices that cover all required capabilities on their own is especially challenging, as the manual partitioning of workload on multiple devices requires extra care. Assume a switch with a total TCAM capacity of N entries and a maximum number of M flow tables. Regardless of potential optimizations, both N and M are constrained numbers and the control plane runs into a bottleneck if more than N rules need to be installed or more than M flow tables are required.

A common way for developers to deal with this problem is creating explicit mitigation strategies for missing capabilities or bottlenecks based on resources from multiple devices. This however, is a redundant, complex and time-consuming task, which leads to additional development effort and costs in the best case, and to feature abandonment and prevention of innovation in the worst. These issues, in return, lead to longer development times and release cycles which adversely affect the adoption of SDN as a mainstream technology.

TableVisor addresses the problem of missing or limited capabilities by creating a pool of multiple physical hardware devices and exposing an emulated single switch with extended capabilities towards the control plane. This emulated device can then leverage the functionality provided by TableVisor to alleviate the mismatch between control plane requirements and data plane capabilities. At the same time, TableVisor is able to translate between different control channel protocols and is thus able to incorporate hardware devices into the network that could otherwise not be controlled centrally.

This functionality not only enables new, more complex use cases in the context of softwarized networks, but primarily enables rapid prototyping during application development and allows researchers without access to often expensive, bleeding-edge hardware to experiment with emulated devices featuring state-of-the-art capabilities. Hence, the main focus of TableVisor is to be used as a tool for application development and research activities, while the deployment in live environments is certainly feasible, as our performance evaluation shows.

The TableVisor concept was originally proposed in [7] and has previously been extended in [8]. This paper extends our previous work on TableVisor to further increase its flexibility and enable use cases beyond hardware accelerated, fully-featured multi-table processing. To this end, we investigate the applicability of our approach in combination with programmable data plane devices and extend the feature-set of TableVisor to enable the translation between different control-channel protocols. We present a mechanism that allows the integration of non-OpenFlow capable devices, such as proprietary P4 hardware, into existing softwarized networks without the need for a major redesign.

We implemented two example use cases, effectively creating a low cost MPLS label edge router based on a programmable P4 device as well as a multi-stage ACL device using a combination of widely available single table OpenFlow switches. Furthermore, we present a detailed measurement study regarding the performance of our approach and discuss its impact on the performance of SDN-enabled networks. We show that our approach induces a constant, minor overhead when used with hardware devices and thereby presents a viable solution to extending device capabilities and rapid prototyping during application development and emulation state-of-the-art devices for research purposes. The main contributions of this paper are:

- Introduction of control channel translation to allow the transparent integration of P4 hardware devices into existing softwarized networks.
- Presentation and implementation of real world examples introducing MPLS support to standard SDN switches as well as realizing multi table processing using widely available single table switches.
- Performance evaluation of the TableVisor approach, proving the viability of the proposed concept.

II. Background and Related Work

Many SDN applications rely on sophisticated packet processing and advanced pipelining. Examples include source address validation [9], d-dimensional packet classification [10], wildcard rule caching [11], controller modularization or hierarchical network management [12]. The available hardware, on the other hand, does not necessarily support all required capabilities. We call this phenomenon a "mismatch" between control plane requirements and data plane capabilities (first introduced in [8]).

There are different approaches to deal with this problem. The easiest solution is to just buy devices that provide the required capabilities or use devices with high overall flexibility. Programmable switches in combination with OpenFlow or P4 are promising candidates here. Another, more realistic, approach is to accept – and maybe even encourage – heterogeneous infrastructures and then hide the heterogeneity with unified and silicon-independent APIs. P4 Runtime is a promising recent development that does exactly that. And finally, there are various works that try to improve flexibility and scalability of switches themselves, e.g., with respect to pipeline processing (see Table I). There are, however, **two fundamental problems** that cannot be solved with just unified APIs or improved switches: the conceptual limits of flexibility and the resource constraints of individual devices.

A. Conceptual Limits of Flexibility

From our view, it is not likely that current (or future) device generations provide sufficient flexibility in the long run. Various enhancements to the OpenFlow protocol and corresponding devices – the last generation so to speak – clearly demonstrate this. [13] extended the match-action abstraction to support autonomous stateful decision making. [14] added a new API to allow autonomous generation of packets. [15] proposed approximation techniques to enable application of otherwise excessive data plane procedures. [16] tackled the problem of slow flow table entry installation. Even more important: completely new control plane requirements may emerge that cannot be realized with a new configuration file, pipeline template or firmware update, e.g., in-network support for distributed machine learning and time-sensitive networking [17]. As a result, it is simply not realistic to control every possible device with a unified API such as P4 Runtime, especially if we are talking about devices with bleeding-edge capabilities often used in the research community.

Instead, we need a way to efficiently deal with different existing control channel protocols. The proxy-layer architecture of TableVisor in conjunction with the new protocol translation concept introduced in this paper is a first step in this direction. The method of transparently processing OpenFlow messages is not particularly new. Similar techniques are used for network virtualization [18], [19], [20], to realize hypervisor functionality [21], to inter-operate with non-SDN legacy network equipment [22] or to transparently deal with flow table limitations [23]. The novelty of our approach is that developers can easily create and deploy their own translation application for every possible control channel protocol, without extensive changes to the network operating system or the control apps, which simplifies rapid prototyping and research work. Furthermore, TableVisor allows the utilization of resources from more than one device, which is discussed in detail in the next section.

B. Constraints of Individual Devices

The second problem – resource and capability constraints of individual devices – is equally important. Programmable switches are, like all other hardware devices, limited to the resources and capabilities of a single device. Control plane requirements going beyond what is provided by a single device cannot be satisfied, even if two cooperating devices could easily fulfill the request. Existing work to alleviate this limitation can be distinguished into two categories: software-based solutions that do not touch the switch hardware and hardware-based solutions. We present prominent related work for both categories and explain how previous work differs from our approach. A brief summary of this analysis can be found in Table I.

1) Software-based Approaches: Most software-based solutions focus on either pipeline flexibility or scalability aspects. We start by elaborating on flexibility by comparing the studies in category C1 in Table I and discuss software-based scalability solutions later on.

Several works try to address the problem of pipeline flexibility. SDFTP [12] introduces software-defined flow table pipelines with an arbitrary number of stages and adaptive table sizes. A special mapping logic is used to embed virtual software tables into the hardware tables of the switch. FlowAdapter [24] is a middle layer between the hardware and software data plane that provides support for multi-stage pipeline processing by properly mapping rules onto existing hardware capabilities. FlowConverter [25] tries to generalize the above ideas and presents an algorithm that can translate between different forwarding pipelines. Table Type Patterns (TTP) [31] for OpenFlow also introduce flexibility by allowing the controller to negotiate pipeline details with hardware switches. ALIEN HAL [32] focuses not directly on pipeline flexibility, but follows similar design principles as

TABLE I
COMPARISON BETWEEN TABLEVISOR AND SELECTED APPROACHES FROM RELATED WORK

			Scope			Features			
	Category	Approach	*Pipeline Flexibility*	*Device Scalability*	*Network Scalability*	*Control Plane Transparency*	*Exploit Shared Resources*	*No Infrastructural Changes*	*Separation of Concerns*
Software	C1	Software-Defined Flow Table Pipeline [12]	✔	✘	✘	✘	✘	✔	✘
Software	C1	FlowAdapter [24]	✔	✘	✘	✔	✘	✔	✘
Software	C1	FlowConverter [25]	✔	✘	✘	✘	✘	✔	✘
Software	C2	Infinite CacheFlow [26]	✘	✔	✘	✔	(✔)	✘	✘
Software	C2	Port Based Capacity Extensions [23]	✘	✔	✘	✔	✔	✔	✘
Software	C2	Palette [27], One Big Switch [28]	✘	(✔)	✔	✘	✔	✔	✘
Hardware	C3	NOSIX [29]	✔	(✔)	✘	✘	✔	✘	✘
Hardware	C3	ESwitch [30], RMT [1]	✔	(✔)	✘	✔	✘	✘	✘
		TableVisor (our approach)	✔	✔	(✔)	✔	✔	✔	✔

FlowAdapter and TTP by using a hardware abstraction layer to realize OpenFlow capabilities on legacy network elements. Furthermore, previously proposed approaches like Frenetic [33] and Pyretic [34] revolve around high level languages for programming collections of network switches. However, these approaches are largely limited to OpenFlow and, like other previously proposed solutions, don't address the limitations imposed by the constraints of singular hardware devices. Instead, these approaches focus on a simplification of the programming interface of SDN-enabled devices.

Because TableVisor is also a software-based approach, there are similarities to the above approaches, primarily with respect to the basic motivation. However, there are three important conceptual differences. (1) Existing solutions for software-based pipeline flexibility are, by design, limited to the resources of a single switch, i.e., the approach can only be used if there is enough free space left in at least one of the hardware tables, which imposes inherent restrictions with respect to scalability. TableVisor copes with this important challenge by combining the resources and capabilities of multiple devices into one emulated device. (2) TableVisor is used in a fully transparent fashion as neither the controller nor the applications have to be modified and the approach can be used out-of-the-box in any OpenFlow based network. While some approaches like FlowAdapter and HAL have similar properties, others sacrifice transparency. SDFTP [12], for example, introduces a new southbound interface for all table operations which requires non-trivial changes in the control plane. (3) TableVisor does not solely focus on flexible pipeline processing but rather considers it as one single use case among a broader set of different applications. The application engine presented in Section III-A can be seen as a generic platform to transparently include different functions. Following this approach, we can combine pipeline flexibility with use cases from other research domains such as TCAM space optimizations, control channel logging or protocol conversion.

Category C2 in Table I is addressing device as well as overall network scalability. Solutions for device scalability try to cope with limited capacities and capabilities of a single device, e.g., by adding a virtual switch with high table capacity [26], [16]. However, this requires infrastructural changes and is associated with a performance degradation for all flows that are forced to use the slow path via the virtual switch. Performance characteristics and limitations of virtual switching were intensively studied in the recent past [35], [36], [37]. Even if only a fraction of the traffic is affected, this might be inapplicable for many scenarios [26]. Others try to achieve similar results by exploiting spare resources of co-located devices [23]. However, unlike TableVisor, these approaches do not consider pipeline processing.

Solutions for overall network scalability usually consider flow tables and TCAM space as a shared resource. Palette [27], DIFANE [38] and One-Big-Switch (OBS) [28] are prominent examples. The general idea of *gathering shared resources under a unified abstraction* is similar to what TableVisor does. However, these solutions have a fundamentally different scope and try to abstract the whole infrastructure (i.e., every switch), while TableVisor is a more localized solution focusing on smaller sets of switches. In addition, the aforementioned solutions introduce policy abstractions that change how networks are used and programmed, which impedes transparency to and compatibility with legacy applications. The high level of abstraction introduced by these changes is a blessing and a curse at the same time. It shields application programmers from low level details but makes it difficult, if not impossible, to realize proper pipelining, because the pipelining itself is not covered by the individual abstractions. Looking at the various use cases that are difficult to realize without explicit, application controlled pipelining [9], [10], [11], [12], we argue that this kind of abstraction is not necessarily a panacea. As a result, we designed TableVisor as a transparent proxy layer without changing the interface that is used for pipelining.

2) Hardware-based Approaches: Category C3 in Table I compares prominent examples of hardware-based solutions. The core idea here is to provide programmable network devices with freely definable packet processing pipelines. The FlexPipe architecture of Intel's FM6000/FM7000 series [39] allows programmable parsing of incoming traffic based on a TCAM/SRAM/MUX structure. Protocol Oblivious Forwarding [40] introduces a generic flow instruction set to make the data plane protocol-oblivious. Reconfigurable Match-Action Table (RMT) [1] proposes a model for re-configurable match tables and enables dynamic, in-field reconfiguration of the data plane. Recently, dRMT [41] introduced a new RMT architecture, in which memory as well as compute resources are disaggregated and moved to a general pool that can be accessed by all pipeline stages over a crossbar. ESwitch [30] proposes a novel architecture that is able to generate efficient machine code for SDN switches based on packet processing templates inspired by the OpenFlow pipeline.

The limitations of programmable switches are twofold. First, they require special hardware currently not available in large quantities. While this may change in the future, it is more likely that such devices will complement existing infrastructures, rather than completely replace them. So the control plane has to either deal with this expected device heterogeneity directly, by differentiating between programmable and non-programmable devices in the applications, or use some kind of abstraction as is provided by TableVisor. Second, even if we assume that every device in the network supports the same degree of programmability, these devices will still be equipped with fixed resources, say a total TCAM capacity of N entries and a maximum number of M flow tables. Existing solutions such as RMT [1] and ESwitch [30] can boost these numbers, so that N and M might be much larger compared to currently used OpenFlow switches. But in the end, they are still constrained numbers and the control plane runs into a bottleneck if an application requires $N + X,\ X > 0$ rules for such a device. TableVisor, on the other hand, allows the dynamic emulation of a device that can access the combined resources of a set of switches, say $N * K$ and $M * K$ if the set consists of K identical switches. This allows for very fine grained scalability by tuning K according to the actual demand in the network. NOSIX [29] is the only hardware-based approach that we are aware of that follows a similar design paradigm by exploiting shared resources. Similar to TableVisor, NOSIX envisions a *lightweight portability layer* for the control plane and can make use of special-purpose tables in different switches. However, in contrast to TableVisor, the approach requires special hardware support and breaks transparency.

III. TableVisor

This section covers the design concept of TableVisor[1], presents available functionality, and concludes with possible use cases that can be realized by applying its features.

[1]https://github.com/lsinfo3/JTableVisor

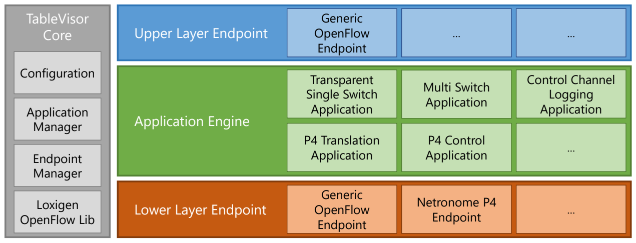


Fig. 1. TableVisor Architecture, including new P4 translation functionality.

A. Architecture and Design

TableVisor has a modular structure in order to allow the addition of extensions and further functionalities in the future. It comprises a central core and three main logical layers, the upper layer endpoint, the application engine, and the lower layer endpoint, as show in Figure 1. The separation between upper and lower layer endpoints as well as the application engine allows a clear separation of responsibilities with regard to the workflow of TableVisor. This is realized through internal Java APIs exposed by the TableVisor Core which enables easy implementation of additional endpoints or application engine modules, compromising novel applications.

The *upper layer endpoint* is responsible for the communication with the controller and handling of protocol specific mechanisms (e.g., keep alive messages). The endpoint parses control protocol messages into a workable data structure that can be processed by the application engine and vice versa. The addition of further upper layer endpoint implementations allows TableVisor to work with control plane instances that are not OpenFlow-capable, e.g., legacy network management systems.

The *application engine* is responsible for passing messages through all loaded applications. An application specifies whether and how a message is processed, e.g., rewriting of table IDs or actions. TableVisor comes with a number of pre-selected applications, including basic applications such as for maintaining the OpenFlow Control Channel, but also applications that change the representation and behavior of the virtual switch.

The applications themselves are structured as an ordered pipeline. Control messages are passed through all loaded applications in a predefined order, which is specified at a global scope during their implementation. Applications can be loaded and unloaded in the configuration file. If an application is loaded, all messages are passed to it, unless blocked by an earlier application. The application order of messages from the upper and the lower layer endpoints are opposed, i.e., the *last* application that is passed by a control message from the upper layer endpoint is traversed *first* by messages from the lower layer, and vice versa. The feature set supported by the current version of the implementation is listed in Table II.

The *lower layer endpoint* maintains the communication paths towards data plane devices. It parses, encodes and decodes messages that are to be sent to or received from

TABLE II
TABLEVISOR APPLICATION ENGINE MODULES.

Transparent Single Switch Application Transparently forwards OpenFlow messages between a single switch and the control plane. Can be used in conjunction with other applications to add functionality, such as the control channel logging application that monitors an OpenFlow control path.
Multi Switch Application Aggregates multiple switches into a single pipeline that is presented towards the controller as a single multi-table switch. Allows the realization of multi-table use cases and exploits the heterogeneity inherent to the landscape of OpenFlow-enabled hardware switches.
Control Channel Logging Application Monitors OpenFlow control messages between the lower and upper layer endpoints. This can be used in conjunction with other applications either for debugging or live monitoring purposes.
P4 Control Application Provides OpenFlow functionality towards the control plane and constructs OpenFlow messages based on statistics and rule sets provided by proprietary P4 management tools.
P4 Translation Application Provides translation functionality between OpenFlow in the control plane and proprietary control protocols in the data plane.

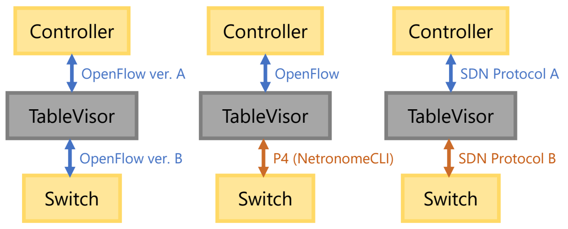


Fig. 2. Illustration of the control channel translation features between specific OpenFlow versions, and generally between different control plane protocols.

the respective data plane devices. This could be OpenFlow messages, implemented by the *Generic OpenFlow Endpoint*, P4 control messages, managed by the *Netronome P4 Endpoint*, or additional novel or existing protocol implementations.

By exploiting upper and lower level endpoints, this architecture can achieve complete transparency towards both data plane as well as control plane. Therefore, it allows the usage of standard controller implementations as well as data plane devices without the need for modifications.

B. Features

The application set of TableVisor is focused around three major feature sets. The main tool used for their implementation is the mapping of global table IDs, as seen by the controller, to local devices and tables, while handling all their intermediate communication. The individual feature domains, as directly applicable by the use cases, are explained in the following.

1) Control Channel Translation: The control channel translation functionality encompasses everything that involves the modification of control messages for compatibility reasons. This includes not only the translation between different protocol families, but also different versions of the same protocol, as detailed in the following.

OpenFlow device heterogeneity. The idea of control channel translation originally evolved around the problem of device heterogeneity in the early OpenFlow hardware implementations, as different switches exhibit different feature sets and device specific behavior with respect to the supported OpenFlow versions, default table numeration, and general conformability with the standard. This problem is most prominent when comparing early implementations of different vendors. TableVisor can be used in such situations to alleviate the problem of protocol mismatches and different expectations between control and data plane. The general instructions of various SDN controllers can be tailored to the needs of specific devices' behavior, and their replies can be generalized to be understood by a standard-conformant software implementation of OpenFlow. This is shown on the left in Figure 2, where TableVisor translates between different OpenFlow versions.

P4 device integration. Going one step further, the new implementation of TableVisor features a semi-automatic translation between the OpenFlow protocol and the P4 running configuration. Therefore, the translation application expects an OpenFlow packet from the upper layer endpoint. As both protocols are based on a similar match-action architecture, the translation engine must merely match the intents contained in the original message to the available table names, action names, and header fields in the P4 program. Note that, as with each translation, the expressiveness of the resulting control channel is limited by the intersection of possible command sets from both languages. In this case, OpenFlow supports a subset of P4 capabilities, hence the level of control is limited by the highest supported OpenFlow version in the controller and the actually deployed P4 program. However, if necessary, unused OpenFlow header fields may be used to address sophisticated headers in P4 programs. In many scenarios, the required operations can be triggered implicitly in the P4 control flow.

TableVisor leverages the existing P4 program in order to learn the respective mappings. Therefore, the program is annotated with the corresponding table IDs, header field names as well as action names and parameters. TableVisor parses these annotations and stores their respective mappings. The following example snipped of a P4 program shows a simple mapping from the OpenFlow table ID `0` to the P4 table name `acl_tbl`.

```
// @TV table 0
table acl_tbl {
   reads {
      ...
```

Similarly, the header field names can be mapped to their OpenFlow counterpart inside the `reads` block of the table. Note that this mapping might be applied locally for this specific table, which enables the use of different mappings in different tables, if desired. Our current implementation applies a global mapping to reduce administrative overhead during configuration. The following annotations map OpenFlow's `in_port` to the corresponding P4 metadata field, and the `ipv4_dst` to our defined destination address header field name.

```
// @TV table 2
table routing_tbl {
   reads {
      // @TV field in_port
      standard_metadata.ingress_port: exact;
      eth.etype: exact;
      // @TV field ipv4_dst
      ipv4.dstAddr: exact;
   }
   actions {
      set_dst_mac;
   }
}
```

Finally, mapping actions requires special care. They do not only contain their own name, but also a list of parameters that may be supplied by the controller at runtime. In addition, OpenFlow allows to execute multiple actions at the same time, while P4 is limited to a single, custom defined action for each table entry. Therefore, a single P4 action `set_dst_mac` is mapped to multiple OpenFlow actions, and vice versa.

```
// @TV action SET_FIELD_ETH_DST ETH_DST=mac
// @TV action GOTO_TABLE_123
action set_dst_mac(mac) {
   modify_field(eth.dst, mac);
}
```

Note here that the `GOTO_TABLE` instruction is called explicitly in OpenFlow, while the table transition is defined separately in the `control` block in the P4 program.

These name mappings are utilized by the translation application to create a JSON string. As our current implementation is designed for the Netronome Agilio CX SmartNIC[2], this JSON string is then passed into the `RTECLI` interface by the respective lower layer endpoint which pushes the new flow rules to the SmartNIC that is associated with it. The entire example mapping, along with further explanation, can be found in the GitHub repository[3].

This implementation is shown in the middle of Figure 2.

Further extensions. Note that the concept can be further extended and generalized to support additional protocols by adding new endpoints, both in the upper and lower layer. With little more effort, new applications could handle the inclusion of additional devices well beyond the match-action abstraction, for example to enable control of hardware accelerated security features in legacy networking devices. These use cases are, however, beyond the scope of this paper.

The general translation between different implementations of SDN protocols is illustrated on the right in Figure 2.

2) Table Capacity Extension: In early deployments of OpenFlow switches, hardware accelerated table capacity was very limited, with some models only allowing a few hundred OpenFlow rules stored in their TCAM[4]. Some use cases require large amounts of flow rules [42], which then need to be handled explicitly by the network administrator or the SDN control plane deployment. With the TableVisor abstraction, multiple hardware tables can be mapped to a single, virtual table and presented in a single device to the control plane. In this scenario, TableVisor would handle dependencies between the different match-action rules, such as implicit header matches due to higher priority flow rules. This concept has been discussed in detail in [8].

[2]https://www.netronome.com/

[3]https://github.com/lsinfo3/JTableVisor/tree/master/example2

[4]https://support.hpe.com/hpsc/doc/public/display?docId=c04217797&lang=en-us&cc=us

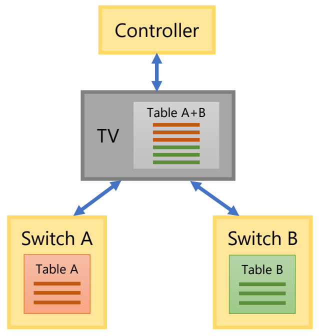


Fig. 3. Illustration of the table capacity extension feature.

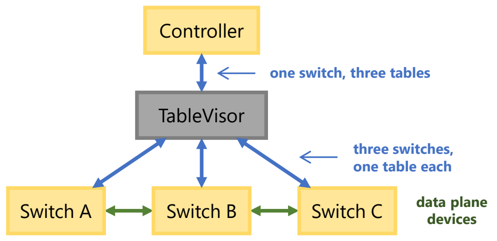


Fig. 4. Illustration of the device aggregation feature.

Figure 3 provides a visual example of the intended use. TableVisor presents a single table hosted by a single device towards the controller. The individual match-action flow rules are split between Switch A and Switch B in order to provide the necessary TCAM space.

3) Device Aggregation: Similarly to the table capacity extension, some features of a single deployed switch are often not sufficient for a task. Simply deploying a "bigger" device would often solve problems with respect to number of ports, number of tables, or even supported features. However, such hardware is not always available, or may be too expensive for the task at hand. TableVisor features a cost-efficient way to aggregate features from multiple networking devices in a single, virtual device. This is especially useful if these devices feature very different characteristics, e.g., a "dumb" forwarding device with many ports and a smart router, VNF, or programmable NIC.

The general approach towards the different aggregation features is shown in Figure 4. The capabilities of multiple devices (Switch A, B, and C) are presented as a single switch with a mix of all of their capabilities, tables, and ports towards the control plane, based on the intended use case.

Port extension. If a single device does not provide sufficient ports to connect to all of its neighbors, but features all of the required functionality, TableVisor can be used to extend the port number by seamlessly integrating another device. The devices would best be connected via dedicated trunk links at higher rates due to the increased amount of traffic between these two switches. Alternatively, multiple ports can be spent for the inter-device communication, depending on the situation.

Extension of number of tables. In many cases, SDN forwarding devices perform multiple actions based on multiple matching criteria. To prevent an exponential explosion of flow rule numbers [43], multiple tables can be used in succession for these tasks. However, not all devices support multiple tables natively. TableVisor can be used to aggregate existing devices and present their tables next to each other in a single, multi table device to the controller. In particular, it would handle the mapping of global table IDs to local, device-specific tables, as well as ensure interoperability, e.g., by translating `GOTO_TABLE` messages to their respective `OUTPUT` actions if the desired table is located on another switch.

Aggregation of features. Finally, not all tables of an SDN switch provide the same capabilities. For example, only a limited number of hardware accelerated rules is often able to push or pop header fields at line rate in the data plane. Sometimes, the required characteristics for a specific task are not simultaneously supported by a single device, e.g., required number of ports and a specific set of actions, possibly provided by sophisticated hardware such as P4 devices like Barefoot's Tofino hardware[5]. In these cases, TableVisor's mapping can be used for a seamless combination of separate devices with the required capabilities. This way, the required characteristics can easily be recreated by available, cheap hardware in a brownfield deployment or for rapid prototyping of new concepts as well as to emulate devices for research purposes.

C. Supported Topologies

This section covers the different underlying topologies the TableVisor approach is able to leverage for its data plane abstraction. Figure 5 shows the four most common pipeline structures. Note that these topologies represent common design patterns. The TableVisor approach is not limited to these types of topologies and can be adapted to the specific use case.

Staged Pipeline. Figure 5a shows the staged pipeline, in which each element provides additional features or TCAM space. All hosts or uplink switches, represented by circles in the figure, are connected to the first stage of the pipeline. From there, packets only need to enter the stage providing the capabilities needed for their specific processing requirements. This minimizes the overhead induced by the abstraction while still enabling more complex use cases. The abstraction is realized by emulating multiple tables, each represented by at least one switch of the pipeline.

Unidirectional Pipeline. An extension of the staged pipeline is shown in Figure 5b. The *unidirectional pipeline* allows hosts or up-link devices to connect to both ends of the pipeline. However, packets may only traverse the pipeline in one direction. A direct communication between network elements connected to different pipeline ends is thus not possible in this scenario. Instead, this emulation type is especially useful in VNF (Virtual Network Function) offloading scenarios that only need to handle unidirectional traffic.

Bidirectional Pipeline. The third extension supported by the TableVisor concept is the bidirectional pipeline illustrated in Figure 5c. This layout allows the connection of hosts or up-link switches to either the first or last switch of the pipeline respectively. Simultaneously, the direction of paths through the pipeline is arbitrary in this case and packets can enter as well as leave the switch aggregate at both ends. This structural layout further increases the capabilities of the emulated switch at the cost of further management complexity. This pipeline type is best used in VNF offloading cases that require bidirectional traffic, e.g., in request-response scenarios.

Circular Pipeline. The final pipeline type supported by TableVisor is the circular pipeline shown in Figure 5d. Thereby, data plane devices are arranged in a ring topology and hosts as well as up-link switches can be connected to every element of the pipeline. This significantly increases the total number of ports available for the emulated switch. Hence, this emulation type can be used to deploy use cases in which a large number of ports is required. The drawback of this setup is the need for internal forwarding rules in order to ensure that traffic still reaches its intended destination. Depending on the specific use case, this may, in the worst case, lead to one forwarding rule per connected host or up-link device, significantly reducing the effective TCAM space available for the application.

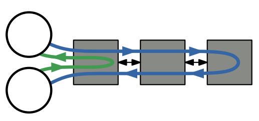
Finally, these different topologies can be used by multiple TableVisor instances in a single network deployment, as shown in Figure 6. In this case, the controller sees five devices in the network, indicated by the control channel connections. The switches connected to each of the respective TableVisor instances are abstracted into a single emulated device. Note that in addition to the TableVisor instances, additional data plane devices can be used without the abstraction layer.

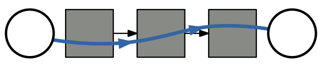
D. Use cases

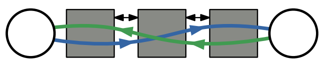
In the following section, we present two specific exemplary use cases that show how the features presented in the Section III-B can be leveraged to design complex scenarios using simple and affordable network components. We start by detailing an access control list (ACL) setup comprised of multiple, non-expensive and highly available single table switches and how the device aggregation feature is used to realize this use case. Furthermore, we show how control channel protocol translation can be used to realize powerful use cases using common network components by describing an MPLS label edge router realized through the incorporation of non-expensive, P4-enabled devices into the data plane.

1) ACL Multi Table Switch: The first example we detail is the realization of an emulated multi-table switch performing access control as well as forwarding. A scenario like this can be especially useful in brownfield deployments in which

[5]https://www.barefootnetworks.com/technology/


(a) Staged Pipeline


(b) Unidirectional Pipeline


(c) Bidirectional Pipeline

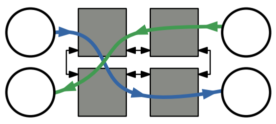

(d) Circular Pipeline

Fig. 5. Available topologies for TableVisor abstraction.

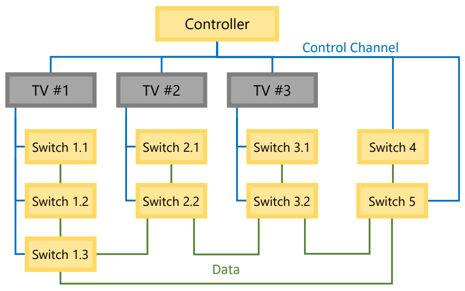


Fig. 6. Schematic depiction of a network using multiple TableVisor instances.

new applications and devices have to co-exist with legacy solutions in both control as well as data plane. The idea here is to combine multiple widely available and affordable single table switches to enable a single, multi table application. In this specific use case, the usage of multiple forwarding tables alleviates the common problem of flow table explosion [43] with legacy SDN devices. This problem essentially describes a combinatorial problem occurring whenever a single or multiple actions need to be performed based on combinations of two or more input values, like ACL rules and MAC addresses. In a single table scenario, all ACL rules would need to be recombined with all possible output MAC addresses, leading to $N \times M$ flow rules in total. When moving to a multi table scenario, the total number of required flow rules to achieve the same functionality is reduced to $N + M$. Thereby, the first table can perform the access control lookups and an independent, second table is used to perform the L2 forwarding task. Figure 7 shows a schematic setup of TableVisor realizing this multi table switch by combining two single table devices.

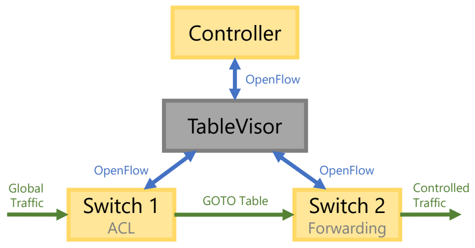


Fig. 7. Schematic ACL setup using two single table devices.

More specifically, we leverage the device aggregation feature discussed in Section III-B, to combine two, potentially heterogeneous, data plane devices into a single, emulated switch featuring two tables. This emulated device is then transparently presented to the controller as a regular hardware device. Hence, neither the devices, nor the controller are required to provide any specialized functionality to be used in this environment. Instead, all involved parties communicate using their version of the OpenFlow protocol. Thanks to the control-channel protocol translation functionality of TableVisor, the two switches as well as the controller can thereby speak different versions of OpenFlow, or even a different protocol all together. At the time of writing, TableVisor supports OpenFlow versions 1.0 and 1.3 as well as the proprietary control API used by Netronome P4 devices used in the example use case in the next section. The processing required for this level of transparency, from control as well as data plane point of view, is fully handled by TableVisor.

Through this message processing performed by TableVisor, the system is able to emulate a multi-table switch towards the control plane without the need for modification of the controller or the involved data plane devices, which in turn increases the reusability of legacy devices in brownfield deployments and allows researchers to quickly prototype control plane applications for which expensive hardware devices would be required, otherwise.

2) P4 Label Edge Router: The second example application we discuss in this work is the realization of an MPLS label edge router using a regular OpenFlow device in combination with a two port P4 PCIe extension card. The goal here is to extend the functionality of a simple, affordable and widely available OpenFlow device by adding the programmability of a single P4 device. By this, we are able to leverage the port capacity of the OpenFlow switch as well as the processing flexibility of the P4 device and combine them into a single, powerful, emulated data plane device towards the controller. The general setup of this application is shown in Figure 8. It is essentially a combination of the *port extension*, *feature aggregation* and *P4 device integration* features described in Section III-B.

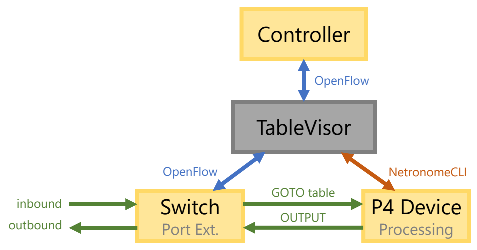


Fig. 8. Schematic MPLS lable edge router setup using a P4 device.

Here, the switch communicates with TableVisor using regular OpenFlow. The P4 device, a Netronome Agilio CX SmartNIC 2x10G, is controlled via the proprietary CLI tool shipped with these kinds of PCIe extension cards. TableVisor thereby handles the translation between the OpenFlow protocol used to connect to the controller and the proprietary control interface of the card. This control channel translation mechanism allows the transparent integration of the P4 device into the data plane without the need for modifications of any part of system. As the SmartNIC only features two ports, the OpenFlow switch is instructed to output all `GotoTable` instructions at one port connected to the NIC and forward packets coming from the other port connected to the NIC. Information required for the forwarding process can be included using the metadata functionality of TableVisor, as described in [8]. The P4 device then performs all the heavy lifting, like pushing and popping MPLS headers, essentially using the OpenFlow switch as a port replicator. Note that the Netronome devices used in this use case feature two 10 GbE ports while the OpenFlow switch only features 1 GbE regular ports and two 10 GbE uplink ports[6]. The implementation of this use case can be found in the GitHub repository accompanying this work[7].

Note that the P4 device in this scenario could also be replaced by a VNF performing the processing tasks. This again simplifies rapid prototyping and seamless integration of complex networking functionality without modification of the control plane in a brownfield deployment.

IV. Performance Evaluation of TableVisor

In order to evaluate the implications of TableVisor in different use cases, we conducted extensive delay measurements with respect to different control plane interactions. We evaluate scenarios in which we measure the controller's interaction with the data plane devices, specifically:

1) how TableVisor influences the overall control channel delay of a bulk of operations, namely FlowMod installations targeted to multiple switches,
2) how delays of individual messages during regular operation are influenced, for example, FlowStatsRequests.

Thereby, the influence of the number of installed FlowMods as well as the number of connected switches is investigated with both software and hardware switches as well as varying controllers.

Figure 9 provides an overview of the experiment setups for the performance comparisons. Here, Figure 9a represents the topology use in a staged pipeline scenario with one to three hardware switches, as well as two hosts[8] for the controller and TableVisor itself. The switches are either connected directly to the controller, or to the TableVisor instance, which acts as a single multi-table switch towards the controller. In order to investigate the impact of horizontal scaling of the pipeline, a similar topology with up to 50 switches has been emulated using Mininet, as shown in Figure 9b.

[6]HP 2920-24G + 2x10G

[7]https://github.com/lsinfo3/JTableVisor

[8]Controller: Intel Xeon X5650 @2.67GHz with 36GB RAM, TableVisor: Intel Xeon D-1548 @2.00GHz with 32GB RAM

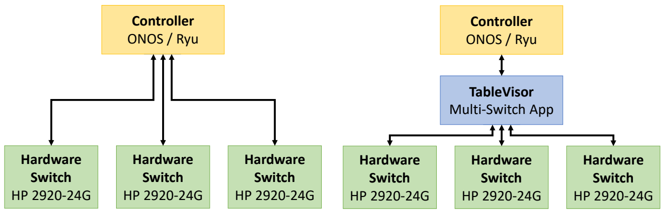
In particular, we measure the FlowMod installation times with and without TableVisor by sending multiple `FlowMod` messages followed by a single `BarrierRequest`. Thereby, without TableVisor, each switch receives its FlowMods directly from the controller. In the scenario with TableVisor, all FlowMods are sent to the TableVisor instance, which then distributes the rules to its underlying hardware devices acting as tables in the emulated switch. The installation time is then given by the time difference between the first `FlowMod` and the last `BarrierReply` as seen on the controller host. This measurement methodology has been evaluated in detail in [44]. For all such experiments, 10 repetitions of each configuration were performed, and all displayed results show the mean installation time with a 95% confidence interval. The flowmods used during the evaluation consist of a match on ethertype 0x800 (IP), the TCP protocol as well as a randomly chosen TCP port. Additionally, the priority of the flowmod is chosen at random between 1 and 255.

For the single-response delay measurements, we captured 30 seconds of ONOS' regular operations after installing the FlowMods. During this time, ONOS sends `FlowStatsRequest` messages to every connected switch every 5 seconds. For each such request, we measure the time difference between the `FlowStatsRequest` and the corresponding `FlowStatsReply` message as perceived by the controller. Note that TableVisor does not aggregate multiple FlowStatsReplies into a single message, but passes each of them separately to the controller by leveraging the `ReplyMore` flag. Based on all such request-reply-delays observed during the 30 second interval, we display their mean values with 95% confidence intervals.

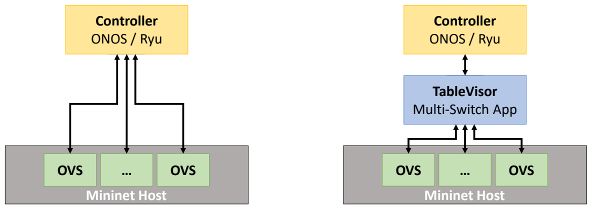
A. *FlowMod Setup Times*

Figure 10 shows the FlowMod installation times with one, two, and three hardware switches, using the Ryu controller on the left, and the ONOS controller on the right. On the x-axis, the number of FlowMods that every switch receives is given, while the y-axis displays the corresponding setup times in seconds. The color indicates the number of switches used in the experiment. Solid lines indicate the bare controller-switch-scenario, and dashed lines refer to the same scenario including TableVisor, as depicted in Figure 9a.

In both controllers' scenarios, the installation times without TableVisor, as shown by the solid lines, are very similar. They range from 0.1 to 1.5 seconds, and the number of connected switches only has a minor impact on the measurements. The steep increments after 200 and 350 FlowMods are expected to originate from the switches' underlying hardware setup, i.e., the time it takes for the switches to process the FlowMods increases with their TCAM utilization [4]. With ONOS, TableVisor causes an additional delay of roughly 0.2 up to 0.25 seconds in the control plane throughout all switch counts and FlowMod numbers. With Ryu, this additional delay shows a slight linear increase with FlowMods after connecting multiple switches. This is due to Ryu generating and sending the messages through a single thread while ONOS uses multiple



(a) Performance overhead measurements with hardware switches.



(b) Mininet scenarios for switch number performance tests.

Fig. 9. Evaluated scenarios and topologies for the performance tests with variable number of flows and switches.

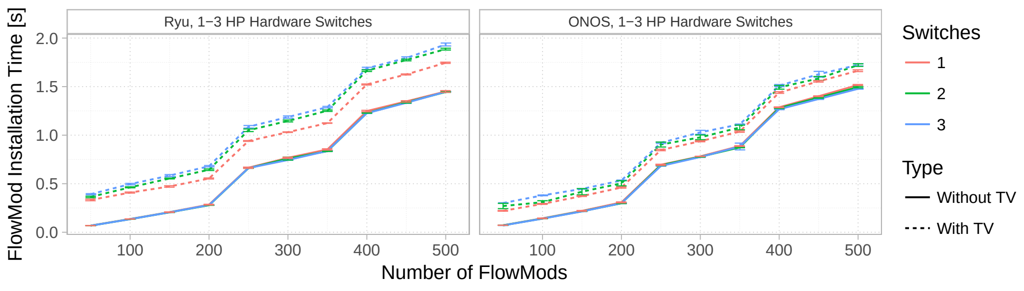


Fig. 10. FlowMod setup times for 2 controllers (Ryu, ONOS), 1 to 3 hardware switches, with and without TableVisor, and 50 to 500 FlowMods.

threads to send messages. Overall, the additional delay caused by the TableVisor proxy layer is dominated by the inherent FlowMod installation times of the hardware switches for higher loads.

The FlowMod installation times obtained from the software scenario in Figure 9b are presented in Figure 11. Hereby, the x-axis shows the number of OVS instances connected to either the controller or TableVisor, and the y-axis contains the respective setup times, as before. The graph is split into groups for each of the both controllers, Ryu and ONOS, and each of the FlowMod counts of 50 and 250, while the color indicates whether the switches were connected directly or TableVisor was used.

When using Ryu, the FlowMod installation times increase linearly with the number of connected devices, both in the 50 and 250 FlowMods case, peaking at 0.25 and 1.25 seconds, respectively. In low-load scenarios, the use of TableVisor causes an additional latency of up to 0.13 seconds. This overhead is decreasing with increasing load on the setup, and after 15 switches with 250 FlowMods, it becomes negligible compared to the bottleneck Ryu introduces. This increment in the performance of TableVisor is likely due to the Java JIT compiler that is able to perform various optimizations at runtime. The results obtained in the ONOS scenarios show a much smaller installation time due to more efficient processing via multi-threading. With 250 FlowMods, ONOS setup times peak at only 0.25 seconds without TableVisor, and rise up to 0.75 seconds with TableVisor. This additional delay is not only caused by the processing time of TableVisor, but also by ONOS itself as it generates and transmits the `FlowMod` messages in a much slower pace when connected to TableVisor, presumably due to its internal multi-threading structure as we only present a single virtual switch to the controller. Note that, when the abstraction of large device numbers is desired, the devices can be split into multiple TableVisor instances. Consider the scenario indicated by the purple line in Figure 11, in which two TableVisor instances each handle 50% of all connected devices. Here, the installation times peak at 0.5 seconds when both instances handle 25 switches, which effectively halves the overhead introduced by our concept in that case.

B. *FlowStats Request-Reply Delay*

Figure 12 presents the delays for the individual FlowStats Requests with up to three hardware switches as well as up to 50 OpenvSwitches. In the hardware measurement in Figure 12a, the x-axis depicts the number of installed flow rules, while the y-axis shows the measured delay between FlowStatsRequest and FlowStatsReply. The latency without TableVisor ranges from roughly 10 ms to 155 ms and is mostly independent of the switch count. The additional delay introduced by TableVisor is steady around 10 ms to 15 ms throughout all investigated FlowMod counts for one and two switches, rising much slower compared to the overall latency. With three switches, this difference peaks at 25 ms with 500 FlowMods, which equals a 16% overhead compared to the pure ONOS case.

Finally, Figure 12b presents the FlowStatsRequest delays with an increasing number of software switches. In this case, the y-axis again shows the measured delay of the request-response pair, while the x-axis shows the number of connected software switches. While the pure ONOS measurements appear to be nearly constant for 50 FlowMods, they increase linearly from 3 ms to 13 ms when the FlowStats Replies contain 250 FlowMods each. The TableVisor measurements

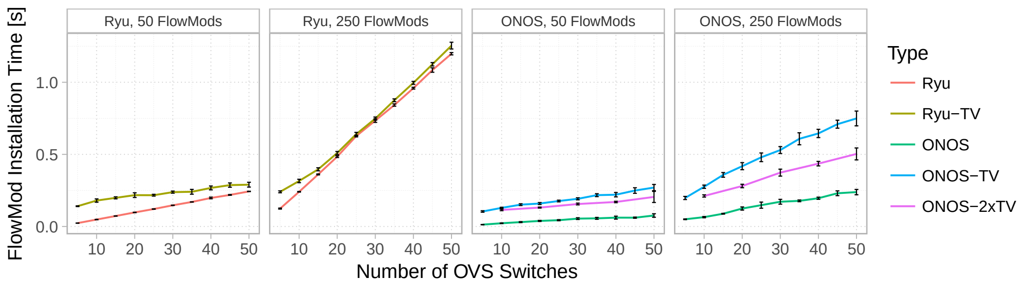
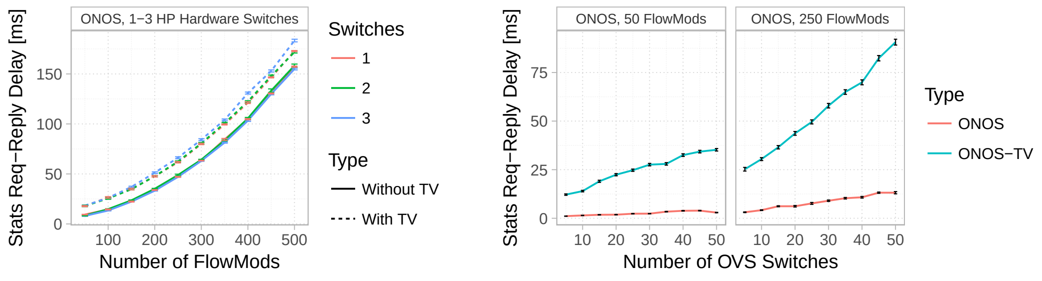


Fig. 11. FlowMod setup times for 2 controllers (Ryu, ONOS), with and without TableVisor, with 50 and 250 FlowMods per switch, and 5 to 50 software switches.



(a) Delays with 1-3 hardware switches.

(b) Delays with 5-50 software switches.

Fig. 12. FlowStats Request-Reply delays for hardware and software switches with the ONOS controller.

show a notable overhead of 12 ms to 34 ms with 50 FlowMods, and 22 ms to 76 ms with 250 FlowMods, respectively, peaking at a mean response time of 89 ms with 50 connected switches and 250 FlowMods. However, it should be noted that these results apply to this specific scenario, and do not only comprise the processing delay of TableVisor. Albeit the mean latency remains quite stable throughout the different samples, the delays of individual switches vary a lot when using TableVisor. Most notably, it should be considered that all software switches are located on the same host in the Mininet scenario. When TableVisor receives the `FlowStatsRequest` message, it immediately sends 50 copies of it towards the switches. However, when ONOS is directly connected to them, the individual requests are spaced out throughout a certain period, allowing the Mininet host to spread the workload and reply to individual requests faster. If we compare the time difference between the first `FlowStatsRequest` and the last `FlowStatsReply`, similarly to the previous FlowMod installation time measurement, we observe 0.175 seconds with TableVisor and 0.202 seconds with ONOS directly connected to 50 switches in the 250 FlowMod scenario. Overall, although individual messages receive an overhead delay from TableVisor during such a bulk update, the total time until the controller receives all updates is even shortened by this approach.

C. Data Plane Overhead

As TableVisor only interacts with the control plane of OpenFlow devices, the above evaluation does not consider data plane performance. Here, we evaluate the latency introduced in the data plane by concatenating multiple data plane devices into a single pipeline. To do so, we measure the end-to-end

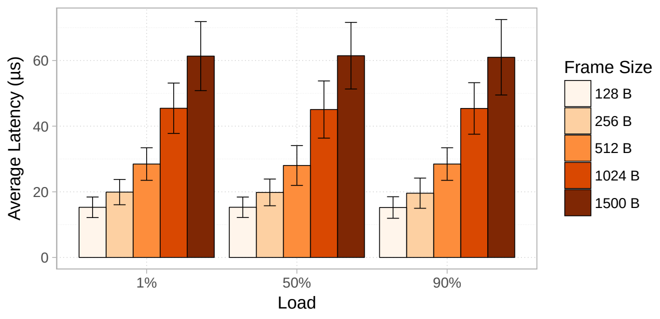


Fig. 13. Data plane delay of an emulated multi-table switch for different frame lengths and load levels.

delay of the emulated pipeline using a Spirent C1 Testcenter as the traffic generator.

Figure 13 shows the average delay in a pipeline comprised of four identical HP 2920-24G switches along the y-axis. Different load levels, in percent relative to the maximum of 1 Gbps, and frame lengths are indicated along the x-axis and by differently colored bars, respectively. The whiskers describe 95% confidence intervals. It can be seen that both the delays and the confidence intervals are independent of the load and are constant for all three load levels. The frame length, on the other hand, has significant influence on the end-to-end delay, which indicates that a large portion of the time is used to write the information to the transmission media while the processing of packets is independent of the packet size.

Note that this evaluation is independent of the TableVisor implementation. This particular scenario is similar to the multi stage ACL deployment presented in Section III-D1 to alleviate shortcomings in the devices' capabilities. Although the setup is achieved by our TableVisor proof of concept, there is no further interaction after preparing the configuration as the data plane traffic does not trigger actions in the control plane.

V. Challenges and Limitations

In this work we propose the concept of data plane device aggregation and provide a proof-of-concept implementation of our approach. In order to detail required considerations and limitations of TableVisor, we discuss the most important points regarding the operation of TableVisor. Furthermore, we explain current limitations of the implementation as well as general limitations of the approach and describe trade-offs that come with the application of TableVisor. Where applicable, we provide potential mechanisms to alleviate the limitations due to the device abstraction.

Performance Characteristics. The performance with respect to additional control plane delay as well as data plane latency when using our approach has been described in the previous section. However, there are some aspects that need to be taken into account when it comes to real world deployments of TableVisor, including research and development scenarios.

Regarding data plane performance, one has to consider the limitations of all devices involved in a TableVisor emulated switch. As traffic needs to traverse multiple devices, the total delay to go through the emulated switch will be the sum of individual device delays. This effectively limits the number of stages used in a specific deployment in order to not exceed the data plane delay requirements. However, as observed in Figure 13, the relevant timescales are in the order of tens of microseconds. Additionally, in the real world, it can be expected that this issue only impacts very specific use cases, as we have shown in Section III-D that the size of aggregates can be kept low, while still able to solve complex use cases. Furthermore, the available bandwidth between physical devices needs to be taken into account as the traffic of multiple input ports may need to be passed on to a second stage. However, depending on the use case and the available hardware devices, the impact of this limitation can be alleviated by mechanisms like providing multiple uplink ports between stages as a trade-off between externally available ports and internal bandwidth capacity.

When it comes to control plane performance, the response time needs to be considered as well as the feature set of emulated devices. The response time of messages is in general limited by the slowest device in the aggregate. However, TableVisor's use of the `SEND_MORE` flag allows modern controllers to start processing replies before all switches answer to a request, resulting in faster control plane updates.

Finally, it needs to be considered that TableVisor is in many cases not a drop-in solution for already deployed networks. Instead, it is well suited to combat heterogeneity of single devices as well as to provide missing functionality by adding a carefully selected set of devices in combination with TableVisor's aggregation features. However, the actual devices to be combined need to be selected carefully as TableVisor allows the aggregation of nearly arbitrary devices. As TableVisor does not currently provide any form of sanity checks or implicit optimization, it is possible to create emulated devices with unexpected behavior, e.g., unevenly sized tables or specialized tables that lack basic features. This needs to be taken into account during deployment and must be avoided either by using a suitable set of switches or worked around on the controller or application side.

Fault Tolerance. When assessing the fault tolerance of a proxy-based abstraction approach, both the failure probability of the proxy itself and the underlying abstracted devices are relevant. As for TableVisor itself, of course, additional components in the system increase the overall failure probability, as it potentially adds another single point of failure to the control channel. To alleviate this issue, TableVisor was designed without the need to track run-time state to enable fast re-initialization in case of software failures. An outage of TableVisor, from both the controller's and the devices' view, appears as a short disconnect of the control channel and no data is lost in this case. Additionally, the stateless nature of TableVisor enables mechanisms like fast fail-over through hot standby to minimize downtime. These mechanisms will be implemented in future extensions of this work. If hosted on similar hardware as the actual SDN controller, TableVisor is subject to the same hardware failure conditions as the controller itself.

As for failures in the data plane, due to the abstraction, problems of individual data plane devices result in the outage of the entire abstracted device chain. This is a limitation of the proposed approach and can not easily be worked around. Potential approaches to alleviate this issue includes the use of backup devices or the dynamic deployment of OVS instances to take over from a failed device.

Scalability. Finally, when it comes to the scalability of the TableVisor approach, multiple aspects that are related to the previously mentioned points have to be taken into account. First, the introduction of TableVisor into the control channel affects the control plane performance of emulated devices. This has been evaluated in detail in Figures 10 and 11. The evaluation has shown that, when using hardware switches, TableVisor induces a constant offset regarding control plane delay. We have also seen that the behavior strongly depends on the controller used. In general, our measurements have shown that TableVisor scales linearly with the number of devices in the evaluated scenarios with up to 50 switches and 250 FlowMods per switch. We assume that this linear scaling holds true for larger scenarios. However, this needs to be verified by means of additional measurements in the future. However, as with the data plane delay mitigation, we assume that in many real world applications, like the examples shown in Section III-D, the number of devices will be limited. Hence, the results shown in Figure 10 should be considered for real world use cases.

For scalability aspects regarding data plane performance as well as the impact of TableVisor on the fault tolerance of a network, we would like to refer the reader to the previous paragraphs covering those aspects in particular.

Overall, it needs to be taken into account that the emulation features provided by TableVisor come with certain trade-offs regarding the control plane and data plane performance, and may also impact other aspects of a network such as resiliency and fault tolerance. Some of these can be worked around using the controller, while others can be limited to a certain extent, e.g. by restricting the number of hardware devices used in a single TableVisor emulated device.

VI. Conclusion

In this paper we present TableVisor, a transparent proxy layer that allows the emulation of hardware-accelerated data plane devices towards a standard SDN controller. On the one hand, this functionality allows the aggregation of devices towards the controller to simplify its view of the network and reduce overhead, e.g., through topology discovery. On the other hand, it enables more powerful and more flexible use cases through the introduction of hardware-accelerated pipeline processing using multiple data plane devices. In this context, we extended the functionality of previous versions by allowing the integration of P4 hardware into an OpenFlow controlled network and introduced further abstraction concepts. These allow the application of TableVisor as a tool during rapid prototyping and the emulation of state-of-the-art devices for research purposes in addition to the realization of new, more complex use cases.

We performed an extensive performance evaluation to investigate the impact of our approach on the control plane performance. Measurements involving hardware devices have revealed that TableVisor introduces a constant additional delay in the control channel that is independent of the respective workloads.

Our investigation has shown that TableVisor is a suitable tool to realize not only complex new use cases using a combination of hardware devices, but also to support the rapid development promised by the SDN paradigm. In constantly changing and developing networks, the high flexibility provided by TableVisor allows SDN application developers, network orchestrators and researchers to realize use cases that face limitations using single, dedicated hardware devices.

Acknowledgment

This work has been performed in the framework of the CELTIC EUREKA project SENDATE-PLANETS, and it is partly funded by the German BMBF. The authors alone are responsible for the content of the paper. The authors would like to thank Simon Raffeck and Simon Englert for the fruitful discussions and their support during the course of this work.

References

[1] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding Metamorphosis: Fast Programmable Match-Action Processing in Hardware for SDN," in *ACM SIGCOMM Computer Communication Review*. ACM, 2013.

[2] A. Lazaris, D. Tahara, X. Huang, E. Li, A. Voellmy, Y. R. Yang, and M. Yu, "Tango: Simplifying SDN Control with Automatic Switch Property Inference, Abstraction, and Optimization," in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 2014.

[3] C. J. Casey, A. Sutton, and A. Sprintson, "tinyNBI: Distilling an API from essential OpenFlow abstractions," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014.

[4] M. Kuźniar, P. Perešíni, and D. Kostić, "What You Need to Know About SDN Flow Tables," in *Passive and Active Measurement*. Springer, 2015.

[5] S. Sezer, S. Scott-Hayward, P. K. Chouhan, B. Fraser, D. Lake, J. Finnegan, N. Viljoen, M. Miller, and N. Rao, "Are We Ready for SDN? Implementation Challenges for Software-Defined Networks," *IEEE Communications Magazine*, vol. 51, no. 7, 2013.

[6] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, "On Scalability of Software-Defined Networking," *IEEE Communications Magazine*, vol. 51, no. 2, 2013.

[7] S. Gebert, M. Jarschel, S. Herrnleben, T. Zinner, and P. Tran-Gia, "TableVisor: An Emulation Layer for Multi-table Open Flow Switches," in *Software Defined Networks (EWSDN) on Fourth European Workshop*. IEEE, 2015.

[8] S. Geissler, S. Herrnleben, R. Bauer, S. Gebert, T. Zinner, and M. Jarschel, "TableVisor 2.0: Towards Full-Featured, Scalable and Hardware-Independent Multi Table Processing," in *Network Softwarization (NetSoft) on IEEE Conference*. IEEE, 2017.

[9] B. Liu, J. Bi, and Y. Zhou, "Source Address Validation in Software Defined Networks," in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[10] C. R. Meiners, A. X. Liu, E. Torng, and J. Patel, "Split: Optimizing Space, Power, and Throughput for TCAM-based Packet Classification Systems," in *Proceedings of the ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*. IEEE Computer Society, 2011.

[11] B. Yan, Y. Xu, H. Xing, K. Xi, and H. J. Chao, "CAB: A Reactive Wildcard Rule Caching System for Software-Defined Networks," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014.

[12] X. Sun, T. E. Ng, and G. Wang, "Software-Defined Flow Table Pipeline," in *Cloud Engineering (IC2E) on IEEE International Conference*, 2015.

[13] G. Bianchi, M. Bonola, A. Capone, and C. Cascone, "OpenState: Programming Platform-independent Stateful OpenFlow Applications Inside the Switch," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, 2014.

[14] R. Bifulco, J. Boite, M. Bouet, and F. Schneider, "Improving SDN with InSPired Switches," in *Proceedings of the Symposium on SDN Research*. ACM, 2016.

[15] N. K. Sharma, A. Kaufmann, T. E. Anderson, A. Krishnamurthy, J. Nelson, and S. Peter, "Evaluating the Power of Flexible Packet Processing for Network Resource Allocation," in *14th USENIX Symposium on Networked Systems Design and Impl. (NSDI)*, 2017.

[16] R. Bifulco and A. Matsiuk, "Towards Scalable SDN Switches: Enabling Faster Flow Table Entries Installation," in *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, 2015.

[17] "Official website of the IEEE 802.1 Time-Sensitive Networking task group," https://1.ieee802.org/tsn/, accessed: 2019-06-18.

[18] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "FlowVisor: A Network Virtualization Layer," *OpenFlow Switch Consortium, Tech. Rep*, vol. 1, 2009.

[19] R. D. Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, "VeRTIGO: Network Virtualization and Beyond," in *European Workshop on Software Defined Networking*. IEEE, 2012.

[20] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "OpenVirteX: Make Your Virtual SDNs Programmable," in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014.

[21] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015, pp. 87–101.

[22] I. Alawe, B. Cousin, O. Thorey, and R. Legouable, "Integration of Legacy Non-SDN Optical ROADMs in a Software Defined Network," in *IEEE International Symposium on Software Defined Systems*, 2016.

[23] R. Bauer and M. Zitterbart, “Port Based Capacity Extensions (PBCEs): Improving SDNs Flow Table Scalability,” in *28th International Teletraffic Congress (ITC 28)*, Wuerzburg, Germany, 2016.
[24] H. Pan, H. Guan, J. Liu, W. Ding, C. Lin, and G. Xie, “The FlowAdapter: Enable Flexible Multi-Table Processing on Legacy Hardware,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*. ACM, 2013.
[25] H. Pan, G. Xie, Z. Li, P. He, and L. Mathy, “FlowConvertor: Enabling Portability of SDN Applications,” in *IEEE INFOCOM 2017 - IEEE Conference on Computer Communications*, 2017.
[26] N. Katta, O. Alipourfard, J. Rexford, and D. Walker, “Infinite CacheFlow in Software-Defined Networks,” in *Proceedings of the third workshop on Hot topics in software defined networking*. ACM, 2014.
[27] Y. Kanizo, D. Hay, and I. Keslassy, “Palette: Distributing Tables in Software-Defined Networks,” in *INFOCOM, Proceedings IEEE*, 2013.
[28] N. Kang, Z. Liu, J. Rexford, and D. Walker, “Optimizing the ”One Big Switch” Abstraction in Software-Defined Networks,” in *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, 2013.
[29] M. Yu, A. Wundsam, and M. Raju, “NOSIX: A Lightweight Portability Layer for the SDN OS,” *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, 2014.
[30] L. Molnár, G. Pongrácz, G. Enyedi, Z. L. Kis, L. Csikor, F. Juhász, A. Kőrösi, and G. Rétvári, “Dataplane Specialization for High-performance OpenFlow Software Switching,” in *Proceedings of the 2016 ACM SIGCOMM Conference*.
[31] The Open Networking Foundation, “OpenFlow Table Type Patterns, Version 1.0,” Aug. 2014.
[32] D. Parniewicz, R. Doriguzzi Corin, L. Ogrodowczyk, M. Rashidi Fard, J. Matias, M. Gerola, V. Fuentes, U. Toseef, A. Zaalouk, B. Belter *et al.*, “Design and Implementation of an OpenFlow Hardware Abstraction Layer,” in *Proceedings of the 2014 ACM SIGCOMM Workshop on Distributed Cloud Computing*.
[33] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: A network programming language,” *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.
[34] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, “Modular sdn programming with pyretic,” *Technical Reprot of USENIX*, 2013.
[35] M. Shahbaz, S. Choi, B. Pfaff, C. Kim, N. Feamster, N. McKeown, and J. Rexford, “PISCES: A Programmable, Protocol-Independent Software Switch,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM ’16, New York, NY, USA.
[36] A. Beifus, D. Raumer, P. Emmerich, T. M. Runge, F. Wohlfart, B. E. Wolfinger, and G. Carle, “A Study of Networking Software Induced Latency,” in *International Conference and Workshops on Networked Systems (NetSys)*. IEEE, 2015.
[37] P. Emmerich, D. Raumer, S. Gallenmüller, F. Wohlfart, and G. Carle, “Throughput and Latency of Virtual Switching with Open vSwitch: A Quantitative Analysis,” *Journal of Network and Systems Management*, vol. 26, no. 2, 2018.
[38] M. Yu, J. Rexford, M. J. Freedman, and J. Wang, “Scalable Flow-Based Networking with DIFANE,” *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 4, 2010.
[39] R. Ozdag, “Intel® Ethernet Switch FM6000 Series - Software Defined Networking,” 2012.
[40] H. Song, “Protocol-Oblivious Forwarding: Unleash the Power of SDN through a Future-Proof Forwarding Plane,” in *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.
[41] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang *et al.*, “dRMT: Disaggregated Programmable Switching,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.
[42] A. X. Liu, C. R. Meiners, and E. Torng, “TCAM Razor: A Systematic Approach Towards Minimizing Packet Classifiers in TCAMs,” *IEEE/ACM Transactions on Networking (TON)*, vol. 18, no. 2, 2010.
[43] The Open Networking Foundation, “The Benefits of Multiple Flow Tables and TTPs,” Feb. 2015.
[44] A. Nguyen-Ngoc, S. Lange, S. Gebert, T. Zinner, P. Tran-Gia, and M. Jarschel, “Performance Evaluation Mechanisms for FlowMod Message Processing in OpenFlow Switches,” in *IEEE Sixth International Conference on Communications and Electronics (ICCE)*, 2016.

Stefan Geissler is working towards his Ph.D at the University of Würzburg, Germany, where he also completed his Master’s degree in 2016. He is a researcher in the ”Next Generation Networks” research group at the Chair of Communication Networks in Würzburg. His research topics include software defined networking and network function virtualization with focus on performance evaluation.

Stefan Herrnleben finished his apprenticeship as IT specialist for system integration in 2006 at the IBM Deutschland GmbH. In 2015 he completed his Bachelor thesis, with the first version of TableVisor as main contribution, at the University of Würzburg, Germany. After finishing his Master’s degree on analysis and optimization of networks in 2017 he is working as doctoral researcher at the Chair of Software Engineering in Würzburg.

Robert Bauer received his Diploma in Computer Science from Karlsruhe Institute of Technology (KIT) in 2014. Since then, he worked as a doctoral researcher at the chair of Prof. Zitterbart. He is involved in several teaching activities related to software-based networking. His research interests focus on communication systems based on Software-Defined Networking and Network Functions Virtualization.

Alexej Grigorjew received his MSc degree in computer science from the University of Würzburg, Germany in 2016. Since then, he has been pursuing his PhD as a member of the Next Generation Networks research group at the Chair of Communication Networks in Würzburg. His research interests are focused on softwarized networks, including SDN, NFV, and programmable data planes.

Thomas Zinner is associate professor at NTNU, Norway. Before, he was visiting professor at TU Berlin and head of the research group on ”Next Generation Networks” at the chair of Communication Networks at the University of Würzburg. He received his diploma in 2006 and Ph.D. in 2012, from University of Würzburg. His research interests cover the performance evaluation of emerging network technologies like SDN/NFV as well as QoE-centric network and service management approaches.

Michael Jarschel is working as a research engineer in the area of Network Softwarization and Connected Driving at Nokia Bell Labs in Munich, Germany. He finished his Ph.D. thesis, titled “An Assessment of Applications and Performance Analysis of Software Defined Networking”, at the University of Würzburg in 2014. His main research interests are in the applicability of SDN and NFV concepts to next generation networks as well as distributed telco cloud technologies and their use cases.