HyperVDP: High-Performance Virtualization of the Programmable Data Plane

Cheng Zhang, Jun Bi, Yu Zhou, Jianping Wu

***Abstract*—With the advent of P4-specific programmable data plane (PDP), network functions (NFs) can be offloaded into the PDP to achieve high performance guaranteed by hardware. Meanwhile, CPU powers consumed by NFs can be released to user applications. However, as more and more NFs can be offloaded, several problems rooted inside the PDP severely hinder it from facilitating this offloading trend. (1) The existing PDP provides the exclusive data plane abstraction where different NFs cannot operate the same data plane. (2) The PDP is hardly able to deploy NFs in a "hitless" manner.**

In this paper, we propose HyperVDP as a high-performance data plane hypervisor to provision non-exclusive abstraction and uninterrupted reconfigurability on the P4-specific PDP. To achieve virtualization, we design several innovative techniques to equally express functions of all programmable elements in the P4-specific PDP. We implement the prototype of HyperVDP on different target platforms, and evaluate different target-based prototypes by comparing with their counterparts. Results show that BMv2-target HyperVDP averagely prevails over its counterpart 2.5x in performance and 4x in resource efficiency. DPDK-target HyperVDP performs comparably to its counterparts while offering virtualization features which neither of its counterparts could provide.

I. Introduction

Based on the newly emerged reconfigurable match-action architecture [3] [4], the Programmable Data Plane (PDP) enables network operators to customize the behaviors of network devices which used to be functionally fixed and proprietary. Accordingly, P4 [5], as a domain specific language, is proposed to provide an easy-to-learn PDP programming abstraction to network operators. As a result, with the help of P4 language and the P4-specific PDP [6], [7], [8], more and more data-plane-affiliative Network Functions (NFs) start to be offloaded into network devices.

Recently, many research proposals including SilkRoad [9], NetCache [10], NOPaxos [11], Dapper [12], Heavy-Hitter Detector [13], [14], [15] and [16], have shown that NFs implemented in the PDP can benefit from both high performance and spare more CPU power for user applications in servers. With the foreseeable flourish of DSLs and programmable devices in the future, more and more PDP suitable NFs could be considered shifting into data planes for purpose of improving network performance and device efficiency.

However, several barriers rooted inside of the PDP severely hinder this promising trend of offloading NFs into PDPs:

(1) **Exclusive data plane abstraction**. The existing PDP presents an exclusive data plane abstraction and can only be operated by one PDP program once deployed. This exclusive abstraction inevitably leads to the consequences that the existing PDP cannot provide isolation among multiple PDP programs and is incapable of meeting the requirements of multi-tenancy scenarios. For example, as more and more NFs can be implemented in the PDP, different tenants in networks may demand various compositions of NFs and configuration policies at each hop along the network path. These NFs and corresponding configurations, due to the diverse demands from tenants, may be dependent, independent or even conflicted. Therefore, it is necessary to provide isolation among PDP programs to support network scenarios with multi-tenancy.

(2) **Interrupted data plane reconfiguration**. Due to the intrinsic constraints of programmable devices, it is hardly possible to dynamically reconfigure the device without interrupting the operation of the data plane. According to [17] and [6], reconfiguration of a programmable network device usually depends on the size and design of the chip ranging from tens of milliseconds to several minutes. Moreover, all device states and table entries will be lost due to this interruption. Therefore, in order to maintain the consistency of network states, it is necessary to provide operators with an uninterrupted reconfigurability when deploying PDP programs.

Virtualization is one of many possible solutions to provide isolation and hitless reconfiguration. In the field of networks, virtualization commonly refers to network virtualization [18], which can be categorized as either external virtualization, combining many networks or parts of networks into a virtual unit [19], or internal virtualization, providing network-like functionality to software containers on a single network server [20]. However, virtualization of the PDP is rather different from the above notion of network virtualization.

In HyperVDP, the notion of virtualizing the PDP shares the similar insights with the virtualization techniques (e.g. virtual machines) adopted in operating systems [21]. Virtualizing the PDP refers to as creating a virtual PDP that can equivalently simulate the behaviors of the physical PDP. Therefore, by HyperVDP, operators can create multiple isolated virtual PDPs on one physical PDP, and dynamically instan-

This work is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (No.61472213). (*Corresponding author: Jun Bi.*)

Jun Bi and Jianping Wu are with Institute for Network Sciences and Cyberspace, Tsinghua University, Department of Computer Science, Tsinghua University, and Beijing National Research Center for Information Science and Technology, and CERNET Network Center, Beijing 100084, China (e-mail: junbi@tsinghua.edu.cn, jianping@cernet.edu.cn).

Cheng Zhang and Yu Zhou are with Institute for Network Sciences and Cyberspace, Tsinghua University, Department of Computer Science, Tsinghua University, and Beijing National Research Center for Information Science and Technology (e-mail: {cheng-zhang13, y-zhou16}@mails.tsinghua.edu.cn).

A previous version of this paper has been published at ICCCN'17 [1]. The source code of HyperVDP can be found at [2]

tiate/migrate/update/delete NFs in each virtual PDP without interrupting the physical PDP. Besides, various NFs are operated in virtual PDPs and can share hardware resources with isolation in scenarios of multi-tenancy. In this way, the PDP can provide both a non-exclusive data plane abstraction and uninterrupted reconfigurability.

In this paper, we propose HyperVDP, a hypervisor with structural designs and well-abstracted models, to fully virtualize a P4-specific PDP while providing high performance and efficient use of data plane resources. Essentially, HyperVDP is a hypervisor that can interpret data plane programs and manage virtual PDPs on top of the physical PDP. This hypervisor is similar with the virtual machine on the "bare-metal" hardware device in operating systems [22]. In HyperVDP, we proposed three novel techniques and concepts, including the abstraction of virtual PDP, control flow sequencing, dynamic stage mapping, to achieve full virtualization. Additionally, we abstracted a model of stage slot to hold any number of stages. In this way, different stages in different programs can be allocated in the same slot to share the hardware resources (e.g. the match-action tables) and improve efficiency. Moreover, techniques of rapid parsing and pipeline bypassing are used to reduce the performance penalty. Hence, HyperVDP can greatly attain the sweet spot between virtualization and performance. Apart from the designs mentioned above, HyperVDP also provides operators with a set of operating models including the stage slot, the virtual PDP, to support a flexible way of composing NFs. Besides, a runtime control platform is designed to compile P4 program and dynamically manage all NFs as well as virtual PDPs without interrupting the physical device.

We implement two prototypes of HyperVDP based on the BMv2 target and the DPDK target respectively. We compare the BMv2-target HyperVDP with Hyper4 (another virtualization work which will be discussed in Section II) under various benchmarks. Results indicate that BMv2-target HyperVDP is averagely 2.5x performance advance of Hyper4 in terms of bandwidth and latency while reducing 4x resource usage. Then we evaluat DPDK-target HyperVDP by comparing with state-of-the-art software switches, PISCES and Open vSwitch. DPDK-target HyperVDP can process 64-byte packets at the speed of 42.14 million packets per second (Mpps) with 4 CPU cores and has a minor throughput penalty when comparing with Open vSwitch at 45.71 Mpps and PISCES at 45.95 Mpps. And for forwarding large packets, HyperVDP can maintain the line rate of 40 Gbps. Meanwhile, DPDK-target HyperVDP can keep the forwarding latency lower than 9 µs with packets of arbitrary size, and above 90% of 64-byte packets can pass through HyperVDP in less than 7 µs.

In this paper, we make the following contributions:

- We design HyperVDP as a hypervisor that fully virtualizes the P4-specific PDP while maintaining high performance and resource efficiency.
- We propose an architecture of a PDP hypervisor and the abstraction of virtual PDP to represent the virtual view of the data plane.
- We propose several novel techniques and creative models including the abstraction of virtual PDP, control flow sequencing, dynamic stage mapping, to facilitate virtualization of the PDP.
- We implement a BMv2-target and a DPDK-target HyperVDP respectively, and evaluate them by comparing with state-of-the-art software counterparts. Results show that HyperVDP can achieve virtualization with a minor performance penalty. We publish the source code of HyperVDP at [2].

In the next section, we will discuss the related works. Then we show the background of P4 language in Section III. In Section IV, we will illustrate the design of HyperVDP and some key techniques. In Section V, we will evaluate BMv2-target HyperVDP and DPDK-target HyperVDP respectively. After that we will discuss the capability and hardware feasibility of HyperVDP in Section VI, and make a conclusion in Section VII.

II. Related Work

An earlier conference paper [1] presented a preliminary design for HyperVDP including sketching the virtual PDP abstraction and presenting ideas for virtualizing P4 language elements on the PDP. Comparing with the previous work, this paper (1) enhances the motivation of offloading data-plane-enabled NFs; (2) implements the dynamic controller supporting instantiation of NFs in the virtual PDP; (3) sets up a fat-tree topology with different NF compositions instantiated on multiple virtual PDPs, then evaluates the performance of HyperVDP in network-wide granularity. Besides, this paper also intensifies the implementation and publishes the source code of HyperVDP.

To the best of our knowledge, Hyper4, is the only research that shares the similar motivation with HyperVDP in virtualizing the P4-specific PDP. Hyper4 [23], demonstrates the feasibility of virtualization of the PDP. In Hyper4, a hypervisor-like program is designed to provide partial virtualization of the P4-specific data plane, so that simple data plane programs can be equivalently emulated in Hyper4. However, comparing with HyperVDP, Hyper4 has several problems as a data plane hypervisor: (1) Hyper4 lacks a structural design and well-abstracted models, including the virtual PDP, dynamic controller, to interpret P4 programs. For example, *parser*, a programmable element, is statically interpreted to multiple resubmitting actions in Hyper4. However, in HyperVDP, we utilize description header to support arbitrary interpretation of parser. (2) Hyper4 merely supports partial virtualization of P4-specific data plane. Several essential P4 language elements such as *if-else* statement with boolean expressions, control flow, are absent from virtualization. Therefore, this defect makes Hyper4 not adaptable for most P4 programs, which weakens the capacity of Hyper4 to support offloading NFs. In contrast, HyperVDP provides a full-virtualization of the programmable data plane including all programming elements and is more general with P4 programs. (3) Hyper4 suffers from a severe performance penalty due to the heavy use of resubmitting actions in the parser and complex inner implementation logic. In HyperVDP, we use rapid parsing along with other techniques to avoid resubmitting actions in the

parser and achieve both comparable high performance and virtualization capability. (4) Hyper4 introduces an excessive usage of hardware resources due to its hard-coded implementation, which also makes it not applicable to programs with arbitrary number of stages. Comparably, HyperVDP utilizes flow sequencing and stage mapping to logically extend the data plane to support data plane programs with arbitrary number of stages.

Other researches related with offloading, e.g. [24], [17], [25], [26], have been designed to provide both high performance and programmability based on the FPGA programmable data plane. Some of them could meet the non-exclusive or uninterrupted demand to a certain extent. However, they either lack of well-abstracted models for virtualization or lack of a hypervisor to uniformly decouple the programs from the physical data planes. For example, in ClickNP, a set of well-defined elements, similar with elements in Click router [27], are implemented as built-in functions in FPGA and can be flexibly composed by operators to achieve complex processing. However, since ClickNP does not take virtualization into consideration, the running NF instance exclusively controls the physical data plane. Beside of researches on data plane implementation, research of NFs placement algorithm is also a significant topic to implement NF offloading. Several NF placement algorithms are proposed in [28], [29], [30], [31], [32], [33] to deploy NFs in different scenarios meeting various requirements. HyperVDP could cooperate with these algorithms to implement device-level or network-level NFs offloading to provide flexible and high performance NFs services.

Authors in [25] proposed a virtualized FPGA data plane letting multiple virtual routers sharing the physical data plane simultaneously to assist network virtualization. It leads to a limited non-exclusive feature. However, the NF in [25] is confined to a fixed router function and the reconfiguration of a new NF will interrupt the data plane. Based on the FPGA devices, [26] took a step forward and designed the virtual data plane with a pipeline of match-action tables as a more general data plane to provide flexibility. The key distinction between our work and [26] is that HyperVDP is devoted to designing a systematic approach to virtualize the protocol-independent and programmable data plane. Moreover, the work [26] mainly focuses on slicing the OpenFlow-like data plane to afford an isolation feature. There is an essential difference between virtualization of a fixed OpenFlow data plane and P4-specific PDP, since a programmable data plane has a more complex logic and hardware access ability than OpenFlow data plane. The OpenFlow-like data plane itself involves with little programmability of data plane behaviors and is simple enough to meet the non-exclusive and non-interrupted feature without any other special design.

III. Background of P4 Language and Motivations of HyperVDP

A. Background of P4 Language

P4, Programming Protocol Independent Packet Processor, is a recently proposed domain specific language for defining the behavior of the data plane. A P4 program mainly defines (1) a parser, containing a collection of protocol formats and corresponding state machines to parse headers; (2) a Direct Acyclic Graph (DAG) of match-action stages, named control flow, to define how packets are processed; and (3) a deparse process to reconstruct packets.

Practically, the workflow of defining a P4 capable device starts with a P4 program. Firstly, operators write and compile a P4 program through a front-end compiler into a high-level intermediate representation known as HLIR [34]. Then a back end-compiler, e.g. p4c-bmv2 [35] or P4-to-EBPF [36], will adapt the program into different targets including FPGA, CPU, etc. Finally, through a runtime controller, such as P4Runtime [37] and P4-ONOS [38], operators can populate table entries into match-action tables on the data plane and make the target device process packets as they are pre-defined. Swapping a new logic into a P4 target should repeat the above procedure, but all network states will be lost due to interruption of the data plane, which might violate the running policies and degrade performance of the whole network.

B. Motivations of HyperVDP

For a forwarding device in network, it is vital to keep forwarding states consistent. However, as stated above, reconfiguring the logic of the programmable data plane inevitably leads to the loss of all forwarding states, which is unacceptable in most practical industrial scenarios. By virtualizing the programmable data plane, HyperVDP can naturally keep all forwarding states consistently tracked and finely managed. Besides, virtualization also enables HyperVDP to support many other useful use-cases including:

- providing networks services for multiple tenants on one single forwarding device with isolated virtual environment.
- deploying network measurement functions such as sketch-based counting [39] in an on-demand way without interfering with any other applications on the programmable data plane.
- composing complex data plane programs dynamically across multiple programmable data planes to support flexible service function chains.

In order to provide full virtualization of P4 specific data plane, several language elements are concerned by HyperVDP: (1) header parser; (2) metadata, including standard metadata and user-defined metadata; (3) registers, meters, and counters, for stateful processing of packets; (4) match-action stages [1]; each stage contains one match-action table; (5) compound action, composed of one or more primitive actions; and (6) control flow, which is used to compose several stages along with boolean expressions indicating stage branching.

IV. Design of HyperVDP

A. Overview

Figure 1 shows the overall architecture of HyperVDP. HyperVDP acts as a hypervisor on top of a physical P4 specific

[1]We will use stage for short in the following paper.

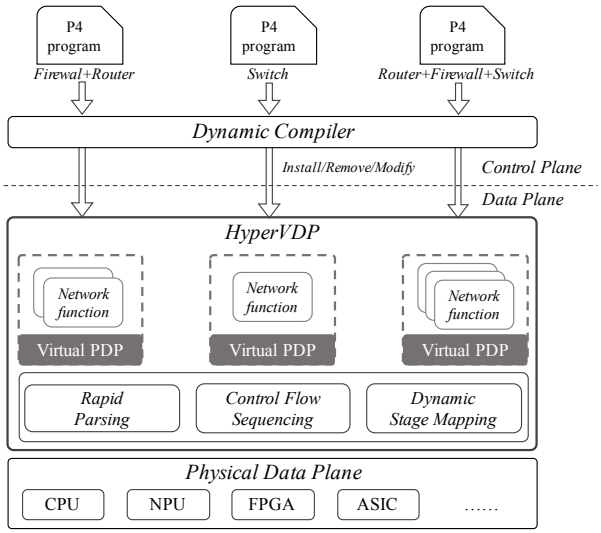


Figure 1: Design overview of HyperVDP.

data plane and provides a virtual view of underlying hardware resources to upper programs. Operators can dynamically configure HyperVDP to create a virtual PDP and instantiate a set of programs on that particular virtual PDP without interfering with other virtual PDPs. In addition, programs can be flexibly instantiated and composed by an operator at runtime on one virtual PDP.

In order to let operators configure HyperVDP from the control plane, we also design a dynamic compiler running in control plane. It can compile and analyze P4 programs created by operators, then transform P4 language elements into data plane models managed by HyperVDP, and dynamically allocate resources such as program id, stage id, stage slot, and populate tables on the virtual PDP.

In HyperVDP, we use three novel techniques and models to achieve high performance as well as full virtualization of P4 language elements. We also design a runtime controller that compiles and manages virtual PDPs dynamically. Through the dynamic controller, operators can flexibly instantiate and deploy NFs in different virtual PDPs with isolation. In the following section, we will provide details of several key models and techniques used in HyperVDP.

B. Rapid Parsing

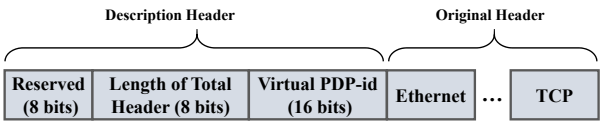


Figure 2: Example of description header.

In a parser of a P4 program, operators can explicitly define an arbitrary structure of a packet header and the corresponding parsing logic of this header. Therefore, virtualizing a header parser means that we must design a general parser being able to parse arbitrary patterns of headers defined by operators. In order to do so, Hyper4 designed a parser with a pipeline of tables and heavily used resubmitting action to parse the header layer by layer, and finally got the header as a whole. However, this design encounters a significant performance penalty since resubmitting is a heavyweight action and each packet suffers from numbers of resubmitting actions.

In order to avoid a severe performance penalty while being able to parse any pattern of packet headers rapidly, we design a 4-byte description header (DH). A DH, as shown in Figure 2, encapsulates the original packet and explicitly identifies a total length of both DH and the original header. It also contains a virtual PDP-id that maps a packet to a virtual PDP. Since in P4 language, a parser can automatically infer the variable length of a member in a header structure by declaring the total length of that header in another member of that header structure. Through this technique, our general parser can view the original header as a whole and infer it from the field in DH without resubmitting packets.

Rapid parsing technique is a trade-off between performance and space. However, we consider it as a reasonable way. Since a general parser should be agnostic to any specific header structure, the key purpose of a parser is to get the header length. So it is more efficient to explicitly identify the length than to parse packet layer by layer to calculate the length.

There is still one concern about implementation of this "extra" DH. Hence, we provide three different ways of implementation to let operators themselves to weigh the balance. First is an overlay method, where a DH can be implemented by MAC-in-MAC, VXLAN, etc., and this method needs an encapsulation mechanism. Second is by modification of destination MAC, which is commonly used within data center networks (e.g. [40]). Third is a transparent way, where each packet will be normally parsed and matched by a unified table to add a DH, then be resubmitted. Meanwhile, operators need to populate some configuration tables when instantiating programs on a virtual PDP.

C. Control Flow Sequencing

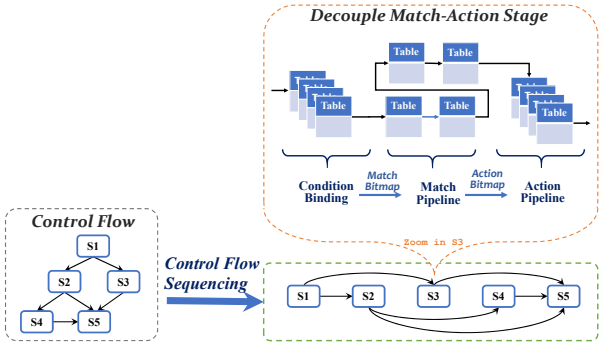


Figure 3: Control flow sequencing.

In a P4 program, a control flow is composed of a set of stages and several boolean expressions. Stages can branch to another stage depending on the result of a boolean expression. Essentially, a control flow is a DAG where each node denotes a stage and each edge represents the branch between stages. So in order to virtualize a control flow, we have to interpret

various DAGs into a unified pattern. Hence, as shown in Figure 3, firstly we decouple a match-action stage into three fixed functional pipelines: condition binding pipeline, match pipeline and action pipeline.

In the condition binding pipeline, we use 4 tables to emulate the boolean expression bound with any particular stage in control flows. This boolean result will decide whether the packet should be processed by the following two pipelines of the current stage, or it should be skipped the following pipelines and branch to the next stage. Correspondingly, an intrinsic metadata will be set to indicate the location of next stage.

In the match pipeline, we classify the match fields in a standard match-action table into three types: packet header, standard metadata and user-defined metadata. Then we assemble the match pipeline with four tables, three of which respectively contains each type of the match field, and the forth one maps the combined match result to an action bitmap. In this way, we avoid using an exceedingly large match field in one table and reduce the TCAM pressure introduced by a large match field. Besides, we also use a match bitmap to indicate whether a table should be executed or skipped in a match pipeline. This pipeline bypassing technique is frequently used by various parts in HyperVDP to further improve performance.

An action bitmap indicates all primitives that should be enforced in a compound action and all primitive actions indicated by an action bitmap will be executed orderly. Besides, we also aggregate all primitive actions, so that an action pipeline demands for a much less of tables to implement. Through an action pipeline, HyperVDP can support a compound action with an arbitrary number of primitives.

After decoupling a match-action stage, secondly we use the topological sorting algorithm to convert a DAG into a uniform linear sequence. Topological sorting can maintain the internal dependency order between stages in a DAG and transform arbitrary DAGs into a uniform pattern. Besides, each stage is allocated with a unique stage id enabling branching from one stage to another.

D. Dynamic Stage Mapping

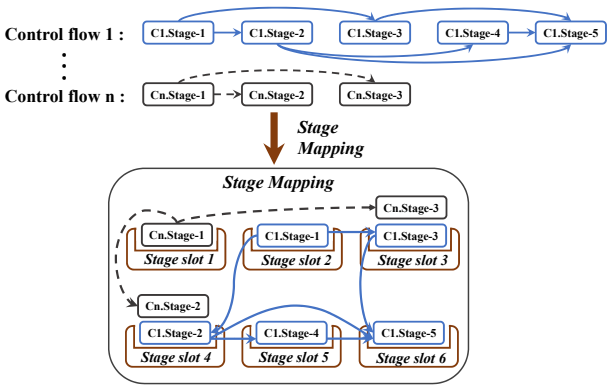


Figure 4: Dynamic stage mapping.

After control flow sequencing, HyperVDP manages to interpret different DAGs into a uniform linear pattern. However, the control flow in a P4 program may contain an arbitrary number of stages. Therefore, HyperVDP faces the challenge of mapping an unpredictable sequence of stages onto the hardware data plane with limited resources. Inspired by the mechanisms used in direct-mapped cache [41], we designed the dynamic stage mapping as shown in Figure 4 to logically extend the capability of the programmable data plane to support arbitrary number of stages.

Firstly, we abstracted a concept of stage slot which can hold any number of stages. Notably, the stage here is a decoupled stage described in Section IV-C. So mapping a sequence of stages into a limited number of stage slots is similar to mapping a virtual memory address to a limited physical memory in the operating system.

Secondly, since each stage is allocated with a unique stage id by the dynamic compiler. Therefore, addressing one stage is by calculating stage id modulo the total number of stage slots on a hypervisor. However, due to the restriction of P4, a stage cannot branch back in order to prevent inner loops. In some cases, if a control flow does have more stages than slots, HyperVDP has no choice but to resubmit packets to resume processing. Definitely, there exists a trade-off between slots and resubmitting actions. It is better to determine the number of slots based on resources of hardware devices and operators' choices.

In this way, we successfully map a flat sequence of stages into a resource-limited hypervisor to support control flows with arbitrary number of stages. Besides, the quantity of physical slots could be adjusted depending on different types of hardware devices. Additionally, stages in different *if-else* conditional branches can be placed in the same slot since they have irrelevant conditional bindings and will not cause any ambiguous table entries when matching packet. A packet will match only one stage from all branching stages in one slot without any conflicts. Therefore, we are on working with a algorithm to further exploit this stage dependency and optimize Hyper+ by permitting slots to hold more stages to fully utilize hardware resources.

Based on the stage mapping technique, HyperVDP not only supports the stage sequence with arbitrary length, but also provides operators with an infrastructural model of the stage slot. Through the stage slot, operators can manage programs on the virtual PDP in a fine-grained way. For example, operators can update (such as add, delete and migrate) the stages dynamically without reprogramming the whole program, and even manipulate stage slots across different physical hardware devices. Consequently, stage slots can be pooled and managed as a unified resource as same as CPUs in a virtualized data center.

E. Dynamic Compiler and Other Language Elements

A dynamic compiler compiles and configures programs on a virtual PDP. The dynamic compiler acquires a P4 program as an input, compiles the program, assigns stage slots for the program, instantiates the virtual PDP as well as other necessary states including program id, stage id etc., and finally populates table entries. In HyperVDP, each packet is

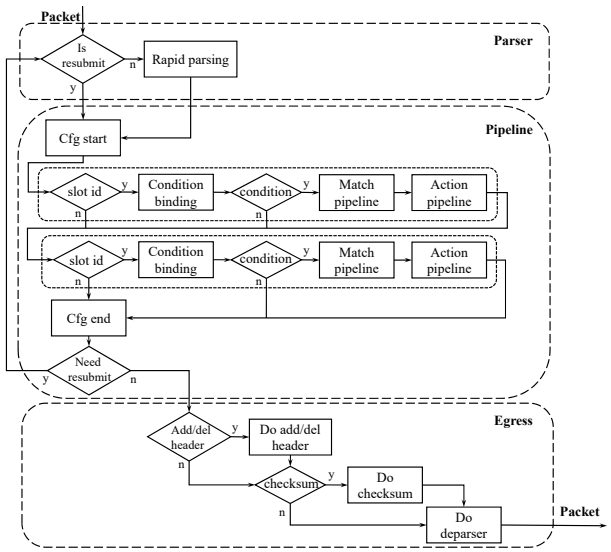


Figure 5: Packet-processing example of HyperVDP with two slots.

	Configuration	Other	One Stage Slot		
			Condition Binding	*Match Pipeline*	*Action Pipeline*
No. of Tables	2	3	4	4	16

Figure 6: Table declaration in HyperVDP.

explicitly tagged with a virtual PDP-id in the description header, and dynamically assigned with a program id and stage id when processed by the virtual PDP according to the tables configured by the dynamic compiler. In the match pipeline of a stage slot, packets will be matched on these ids to enforce isolation and flexible composition of programs.

Figure 5 shows the processing flow of HyperVDP with two stage slots. There are two configuration tables, one is for setting the address of first stage for each packet at the beginning of the pipeline; the other is for resubmitting the packet back to the ingress pipeline in case of insufficient slots at the end of the pipeline. From Figure 6, we can see that one stage slot totally contains 24 tables, in the action pipeline, we optimize HyperVDP to support all P4 language primitives with just 16 tables. Besides, other three tables are used in the egress pipeline of HyperVDP to implement actions including checksum and header modification. Actually, the number of stage slots depends on the physical resources of a hardware device where a high-end device always supports more slots than a low-end device.

The dynamic compiler provides operators with several models to manage programs on a virtual PDP. First is a stage, which is functionally identical to a match-action stage in P4. Stages can be flexibly instantiated in a stage slot or composed to achieve complex processing. Second is a program, which denotes an independent network function such as a *Router*. Different programs can also be flexibly composed or instantiated on a virtual PDP. Virtualization of other P4 language elements, including meters, counters, registers, are implemented in a reserved way. Due to space cause, they will not be covered in detail.

V. Evaluation

Dimension. The evaluation of HyperVDP contains four dimensions. (1) In Section V-A, we compare HyperVDP with Hyper4 in terms of ***resource usage*** including declared tables, reserved metadata and etc. (2) In Section V-B, to evaluate the ***performance of DPDK-target HyperVDP***, we build a testbed and conduct several experiments on DPDK-target HyperVDP, PISCES and Open vSwitch in terms of micro-benchmarks as well as macro-benchmarks. (3) In Section V-C, we design seven test policies where each policy demands different composition of NFs. Based on the policies, we evaluate the ***performance of BMv2-target HyperVDP*** by comparing with Hyper4 and native P4-specific data plane. (4) In Section V-D, to evaluate the ***performance of HyperVDP in network-wide granularity***, we build a testbed of fat-tree topology by BMv2 and measure the performance of HyperVDP. Notably, this experiment also demonstrates the *non-exclusive abstraction* and *uninterrupted reconfigurability* provided by HyperVDP.

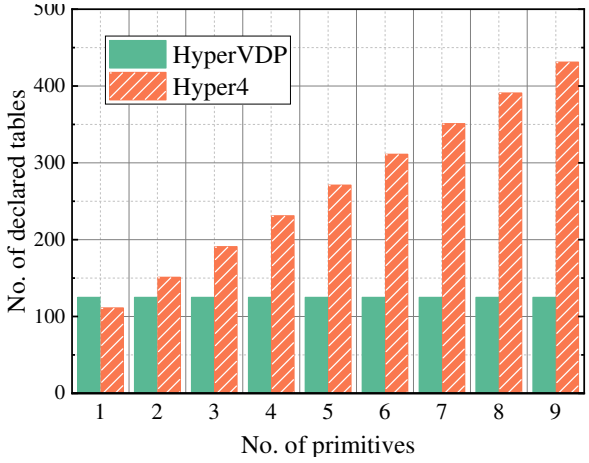
Implementation. The implementation of HyperVDP contain three components. (1) We utilize 3000 lines of P4 code to implement HyperVDP on the BMv2 target. (2) In order to further exploit the feasibility and performance penalty of HyperVDP, we also implemented HyperVDP on the Intel DPDK [42] platform with 8000 lines of C code and evaluated DPDK-target HyperVDP by comparing with Open vSwitch [43] and PISCES [44], which are state-of-the-art software switches, in terms of several benchmarks. (3) Besides, we implement the dynamic compiler for the BMv2-target HyperVDP based on the P4Runtime platform [37] and P4 HLIR abstraction [34] with 6000 lines of C++ code.

Setup. DPDK-target HyperVDP and BMv2-target HyperVDP both run on the DELL R730 PowerEdge servers with 2×4 Intel E5-2620 2.40Ghz cores and 64GB memory. Moreover, the DPDK runtime environment is connected to Intel 82599 NICs [45], each of which is quipped with two 10G ports. Besides, we employ iPerf [46] and MoonGen [47] to test these two HyperVDP targets in terms of throughput and delay. MoonGen is a DPDK-based high-speed packet generator that can provide 64-byte packets on a single CPU core at the speed of 10 Gbps.

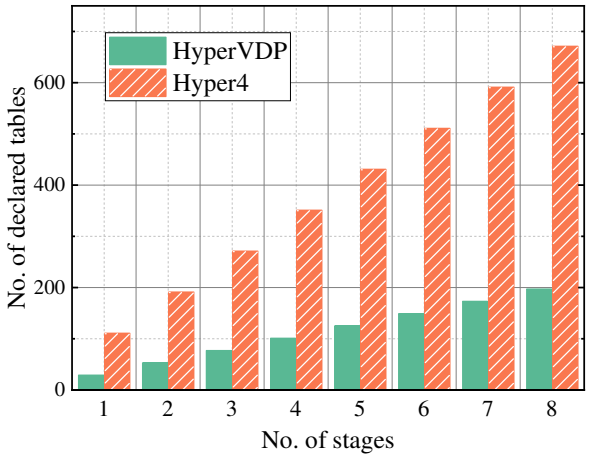
A. Resource Usage

We analyze the resource usage of HyperVDP in terms of three metrics. The first one is *table declaration* of a hypervisor. This benchmark denotes how many tables are required in the hypervisor. Since hardware devices have their memory resource limitations, a hypervisor should always declare resources as less as possible in order to be more efficient and deployable. The second one is *metadata declaration*. It involves the width and type of match fields in a table, and has an impact on memory allocation and TCAM usage. For instance, for the RMT [3] architecture, each stage merely has a limited number of physical memory and TCAM resources for table matching. The last one is *table usage at runtime*. This benchmark identifies how many tables are used when

executing an NF in a hypervisor. Using of more tables in the pipeline prolongs prolong the processing delay, which is proved by following experiments.. Thus, a hypervisor should be optimized to save memory for table matching and run management logic in high efficiency.



(a) Table declaration with different primitives of 5 stages.



(b) Table declaration with different stages of 9 primitives.

Figure 7: Comparison of table declaration.

1) ***Table declaration****:* Hyper4 hard-codes the implementation of a stage pipeline, hence the number of table declaration is closely coupled with the number of stages hard-coded as well as the maximum number of primitives permitted in one compound action. However, table declaration in HyperVDP depends on the number of slots configured by operators and is irrelevant with above two factors. In order to make a horizontal comparison with Hyper4 in terms of table declaration, we also hard-code these two factors respectively in HyperVDP.

In Figure 7(a), Hyper4 and HyperVDP both hard-code 5 stages (5 slots in HyperVDP). As the maximum number of primitives permitted in one compound action increases, Hyper4 suffers from a linear growth of table declaration, while HyperVDP still remains the same. In Figure 7(b), both Hyper4 and HyperVDP set the maximum number of primitives to 9. We can see that in the case of 8 slots, HyperVDP declares only 197 tables while Hyper4 needs more than 650 tables.

Platforms	Metadata (bits)			
	Header	*User*	*Control*	*Total*
HyperVDP	800	256	312	1368
Hyper4	800	256	2256	3312

Figure 8: Comparison of metadata Declaration.

2) ***Metadata declaration****:* Figure 8 shows metadata declaration in Hyper4 and HyperVDP. Due to the fact that a hypervisor cannot infer the length of a packet header until it starts to process the packet. Hence, both Hyper4 and HyperVDP reserve 800 bits (far surpass the common packet header length) to hold a packet header, and 256 bits for user-defined metadata (enough for most programs). Benefiting from a structural design and simple inner control logic, HyperVDP only needs 312 bits of metadata for intrinsic control, while Hyper4 increases over sixfold and needs as many as 2256 bits.

NFs	Native P4	Hyper4	HyperVDP
Switch	2	13	5
Firewall	3	22	8
Router	4	28	16
ARP Proxy	4	48	10

Figure 9: Table usage at runtime for different programs.

3) ***Table usage at runtime*** *:* Figure 9 shows the number of tables used at runtime in HyperVDP and Hyper4. In the experiments, we implement four NFs and test them independently. As *switch* is shown, packets in a native P4 switch flow through two match-action tables, one is a source MAC table and the other is a destination MAC table. Accordingly in Hyper4, 13 tables are required to emulate the *switch*. While in HyperVDP, only 5 tables are enough, including one configuration table at the beginning, two match tables and two action tables. The *L2 switch* in HyperVDP contains 2 slots and each slot contains both match table and action table, therefore, collectively four tables reside in 2 slots. Since a *L2 switch* has no condition binding and has only one type of match field, it can be optimized to bypass other irrelevant tables at runtime.

In the *Router* and *ARP proxy*, HyperVDP uses 16 and 10 tables respectively. Extra tables in the *Router* are used to fulfill complex operations such as TTL validation check, TTL modification and checksum recalculation. Comparing HyperVDP with Hyper4, improvement is especially obvious when dealing with complicated programs. Generally, HyperVDP reduces 2x to 4x of table usage in runtime comparing with Hyper4.

B. Evaluation of DPDK-target HyperVDP

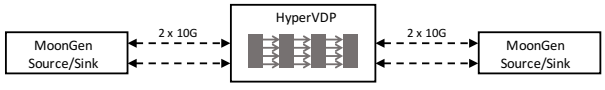


Figure 10: Testbed topology.

1) ***DPDK Testbed****:* Figure 10 shows the testbed for evaluating the performance of DPDK-target HyperVDP. The DPDK

testbed contains three servers, two of which are running MoonGen packet generators and receivers. The software switches and HyperVDP are running on the middle server which has four 10G NIC ports directly connected to the packet generators and receivers through optical fibers. We can use the testbed to generate customized packets at 40 Gbps for the sake of evaluating the forwarding performance of HyperVDP against Open vSwitch and PISCES.

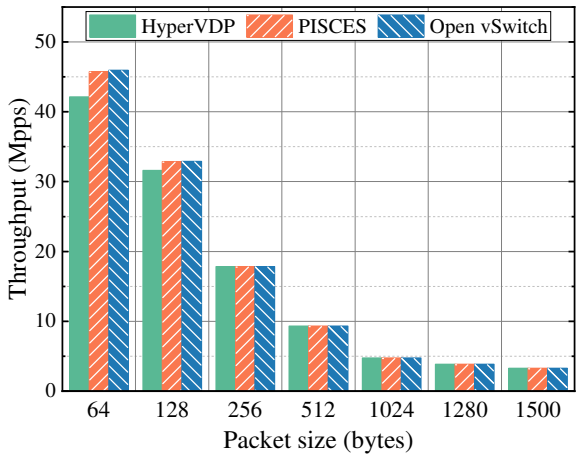
Open vSwitch has multiple data plane drivers, including the kernel driver and the DPDK driver. Due to the fact that HyperVDP is built on DPDK, we choose the DPDK-based Open vSwitch as a high performance software counterpart in this paper. PISCES is a programmable and protocol-independent software switch built on Open vSwitch. PISCES could compile P4 programs into Open vSwitch source code with a minor performance penalty, and PISCES provides an efficient and expressive approach to define the behavior of the software switch. So it is appropriate to view PISCES as a high performance P4-based software switch.

We tested the end-to-end performance of DPDK-target HyperVDP, PISCES, and Open vSwitch in terms of throughput and delay. All DPDK switches in our benchmark were set up to implement the *L2 switch* with four poll-mode threads running on independent CPU cores over the same CPU socket. Apart from the macro benchmark experimentation, a micro benchmark experimentation was also implemented to further understand the performance bottleneck of DPDK-target HyperVDP. We utilized the time-stamp counter library of DPDK, and measured the number of CPU cycles consumed by inner elements of HyperVDP.

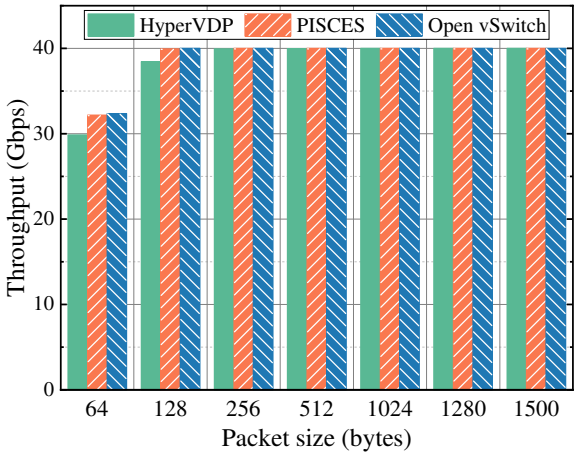
*2) **Macro benchmark**:* The MoonGen packet generators can create the specific size of packets at the full speed of NICs. As for forwarding delay, sample packets with time stamps will be sent to measure the end-to-end delay of software switches. MoonGen will pour the background traffic containing 64-byte packets at the speed of 4 Gbps into the testbed to populate queues in software switches. Meanwhile, sample packets of different sizes are generated to explore the impact of the packet size on the end-to-end delay.

Throughput of three switches in different measuring metrics is shown in Figure 11(a) and Figure 11(b), which illustrate that the DPDK-target HyperVDP implementation has a minor performance penalty while introducing virtualization onto a PDP. In this experiments, the test bed generates data traffic at 40 Gbps constantly to inspect the processing capability of the data plane. We execute each test case for 10 times and get the steady average results in Figure 11(a) and Figure 11(b). We can see from both measuring metrics that for small packet size, there exists minor difference of processing ability among three switches, while for large packets all the three switches can maintain a line-rate throughput. As shown in Figure 11(a), HyperVDP could process packets with 64 bytes at a speed of 42.14 Mpps, meanwhile Open vSwitch and PISCES perform a little better and can forward 64-byte packets at 45.71 Mpps and 45.95 Mpps respectively. HyperVDP introduces a throughput penalty of about 7% in terms of processing small packets.

Previous researches including PISCES and Open vSwitch have shown that the flow cache is of great significance in



(a) Throughput in packets per second (Mpps).



(b) Throughput in Gigabits per second (Gbps).

Figure 11: Throughput comparison.

enhancing forwarding performance, while the existing prototype of DPDK-target HyperVDP does not use any optimization technique of flow caching to reduce the throughput penalty. In spite of the throughput penalty comparing with PISCES and Open vSwitch, we deem it worthy to introduce virtualization into the data plane at the expense of a minor performance penalty.

Delay of packets is another concerned aspect in the benchmarks of DPDK-target HyperVDP. The mean latency is illustrated in Figure 12. As it can be seen, the latency of packets increases slightly with the packet size. Open vSwitch and PISCES process packets with latencies ranging from 4.42 µs to 7.24 µs. HyperVDP can process a 64-byte packet in 6.28 µs on average.

Both of PISCES and Open vSwitch adopt the run-to-completion model as their packet processing model. In the run-to-completion model, each packet or each batch of packets in the data path is assigned a single thread that completes the full functionality of packet processing at one time. Nevertheless, DPDK-target HyperVDP is based on the software pipeline model, which amortizes the pipeline into individual elements running on separated worker threads. Due to the fragmentation of the switch pipeline and extensive use of FIFO queues, the latency of a packet traversing the pipeline of HyperVDP is

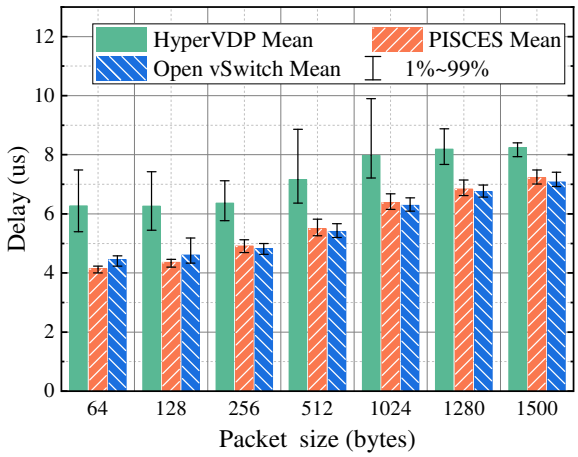


Figure 12: Latency comparison.

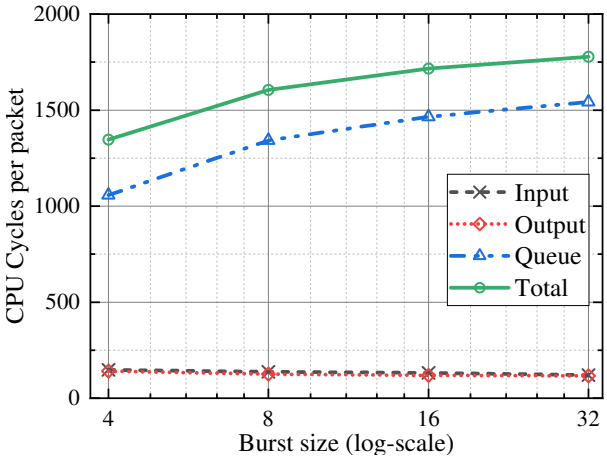


Figure 13: CPU cycle consumption.

theoretically higher than PISCES and Open vSwitch.

As aforementioned, the goal of this evaluation is to inspect the performance penalty introduced by HyperVDP accurately. We can infer from the results that the performance penalty of HyperVDP is acceptable when considering the up-and-coming features provided by virtualization of a programmable data plane.

*3) **Micro benchmark**:* To understand bottlenecks of DPDK-target HyperVDP in depth, we make further efforts to measure CPU cycles consumed by inner elements of HyperVDP. The prototype of HyperVDP has several processing elements which can repeatedly process batches of packets. (1) The first element encountered by ingress packets in the pipeline is the input node. It maintains RX queues of all physical or virtual ports, from which the input node can fetch the incoming packet bursts. Apart from receiving packets, the input node should also configure the packet metadata and then distribute pointers of packets into the queues of target elements. (2) The general nodes, representing the match-action tables, implement customized match fields and action primitives. Multiple nodes can share one CPU core to improve the utilization of the computational resources while causing a sacrifice in performance. (3) The last kind of element is the conditional node which can process a conditional statement. Through this node, HyperVDP could implement arbitrary conditional semantics of P4 language or other data plane DSLs.

As stated in Section V-B2, the increase of delay is partially introduced by the queue model implemented in DPDK-target HyperVDP. Moreover, since elements in HyperVDP are designed to process batches of packets, the size of a packet batch might also has an impact on delay. Thus, we inspected the influence of a queue on delay in micro benchmark by changing the batch size of a queue. In the tests, the background traffic of 64-byte packets at 1 Mpps is used to populate queues of HyperVDP. We respectively measured CPU cycles consumed by the input node, the general node and the FIFO queue and the total. Measurement results of CPU cycles consumed by different elements are plotted in Figure 13. Additionally, we also measured CPU cycles consumed by different selected actions in HyperVDP. We will illustrate the results separately as follows.

Per-packet cycle counts of the pipeline (*Total*) increase as the batch size of the queue becomes larger. As it can be seen from Figure 13, the queue (*Queue*) consumes a large part of total cycles, and the ratio is about 86.74% when there are as many as 32 packets per batch. In contrast to the queue, cycles of the input node (*Input*) and the general node (*Node*) remain the same when the burst size increases. In fact, cycles per packet of an input node and a general node decrease by several cycles with the increase of batch sizes, because the packet batch could amortize the cost of processing, e.g. memory accesses, function calls, over packets in a batch. Thus, the delay penalty partially originates from the slightly excessive use of the queue in HyperVDP. And we also have executed some straightforward experiments on the end-to-end delay and throughput with different batch sizes. The results do verify above conjectures and reveal that the smaller batch size leads to lower delay while causing a decline in throughput.

Cycle counts of primitive actions are concluded briefly as below. As a compound action can have more than one primitive action, the cycles consumed by the compound action will increase with the length of the action list, which causes another potential performance bottleneck. Thus, we tracked the performance of P4 primitive actions in general nodes. MODIFY_FIELD consumes about 33 CPU cycles per packet on average, when it's appointed to modify L2 source address. MODIFY_FIELD_WITH_MASK is more complex than MODIFY_FIELD, and can modify discontinuous fields with a mask. The number of cycles consumed by MODIFY_FIELD_WITH_MASK is about 39. ADD_HEADER performs a sophisticated action adding a predefined header into the rigid header stack, which requires movement of memory blocks. Thus, ADD_HEADER consumes as many as 51 cycles. REMOVE_HEADER only needs about 45 cycles.

C. Evaluation of BMv2-target HyperVDP

For the performance evaluation of BMv2-target HyperVDP, we firstly implement six network functions as shown in Figure 14(a). Then we design seven different policies demanding various NF compositions as shown in Figure 14(b). For example,

Name	Function Descriptions
MAC Learn	Learn source MAC addresses and input ports.
L2_SW	Forward packets based on destination MAC addresses.
L3_SW	Forward packets based on destination IPv4 addressees.
IP_SG	IPv4 source guard.
FW	Filter packets according to layer-3 and layer-4 information.
QoS	Set the queue id.

(a) NFs and descriptions of each NF.

Name	NF Compositions
Pol. 1	L2_SW
Pol. 2	MAC Learn → L2_SW
Pol. 3	MAC Learn → QoS
Pol. 4	L3_SW
Pol. 5	L3_SW → FW
Pol. 6	L3_SW → IP_SG → FW
Pol. 7	MAC Learn → L3_SW → IP_SG → FW → QoS

(b) Test cases composed of different NFs.

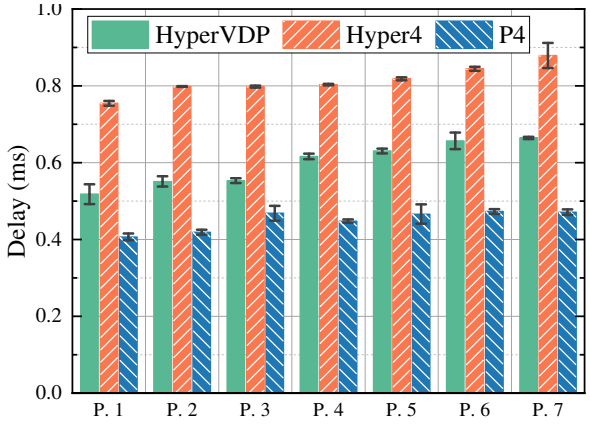
Figure 14: NFs and test policies for evaluation of BMv2-target HyperVDP.

packets that belong to the composition *Pol. 6* should firstly traverse L3_SW, then IP_SG, and FW at last. We use the NF compositions to respectively evaluate the throughput and delay of BMv2-target HyperVDP, Hyper4 and native P4 (all the three platforms run on the BMv2 software switch). For native P4, we directly implement the NF compositions exactly as demanded by each policy in the table. Consequently, for experiments conducted on native P4, these NF compositions are statically configured into the BMv2 target without any virtualization feature. However, for HyperVDP, those NF compositions can be dynamically offloaded, modified and removed according to high-level network intents.

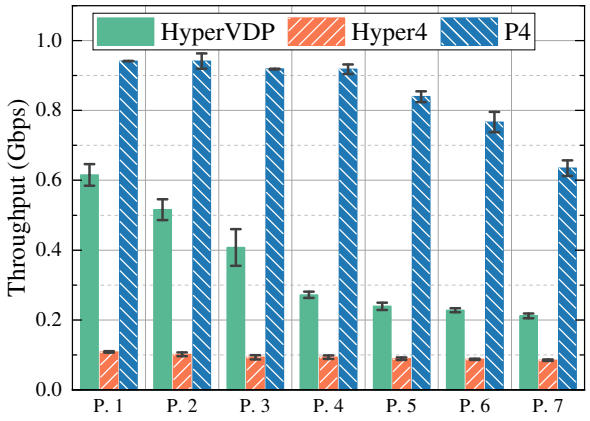
Delay of the three platforms increase gradually. As can be seen from Figure 15(a), the increasing delay of Hyper4 ranges from 70.4% (*Pol. 3*) to 90.6% (*Pol. 2*) when comparing with native P4. For HyperVDP, the increasing delay remains below 41.0% (*Pol. 7*), and the minimal increasing delay is only 27.4% (*Pol. 1*). When making a comparison between HyperVDP and Hyper4, we can see that HyperVDP acquires a decreasing delay by 26.5% on average.

Throughput of the three platforms is shown in Figure 15(b). Hyper4 incurs a large throughput overhead which can be a decrease of 89.8%. On average of the seven test cases, the decreasing throughput of Hyper4 is 88.9%. For HyperVDP, the decreasing throughput ranges from 34.6% to 70.3%, which is much smaller than Hyper4. In the best cases, HyperVDP can achieve an increasing throughput by 466.3% comparing with Hyper4 in *Pol. 1*

Overall, HyperVDP not only provides richer semantics to support offloading more NFs to the PDP, but also maintains the performance overhead in an acceptable range which enables the data plane hypervisor to provide high-performance and virtualized services to various traffic flows. In the next section, we



(a) Delay comparison.



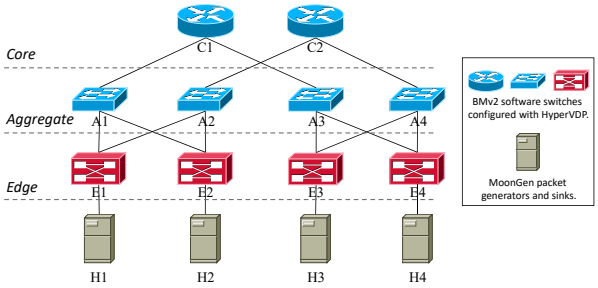
(b) Throughput comparison.

Figure 15: Performance of BMv2-target HyperVDP.

will elaborate the performance of HyperVDP in network-wide granularity, where different NF compositions are instantiated for different virtual PDPs on various programmable devices in network-wide granularity.

D. Evaluation of HyperVDP in Network-Wide Granularity

To elaborate the performance of HyperVDP in network-wide granularity, we employ ten BMv2 software switches to build a two-pod (i.e., *H1* and *H2* are in one pod, *H3* and *H4* are in another pod.) fat-tree topology shown in Figure 16(a). In the topology, we connect one server to each edge switch, and these servers act as not only packet generators but also packet sinks. Flows are categorized into two types: one is inter-pod (e.g., from *H1* to *H3*), another is intra-pod (e.g., from *H1* to *H2*). The NF compositions of *Edge(IN)* are applied to the flows coming from the servers, while the NF compositions of *Edge(OUT)* are designed for the flows going to the servers. For every flow in each device along the forwarding path, different NF compositions shown in Figure 16(b) are instantiated in different isolated virtual PDPs on the hardware device. For example, the intra-pod flow emitted by *H3* will be processed by composition of *MAC Learn*, *L2_SW* and *QoS* at *E3* in one virtual PDP context, while the intra-pod flow destined to *H3*



(a) Fat-tree topology.

Flows	Positions	NF Compositions
Inter-pod	Edge (IN)	L2_SW → IP_SG
	Edge (OUT)	L2_SW
	Aggregate	L2_SW
	Core	L3_SW → FW → QoS
Intra-pod	Edge (IN)	MAC Learn → L2_SW → IP_SG → QoS
	Edge (OUT)	MAC Learn → L2_SW → QoS
	Aggregate	MAC Learn → L2_SW → QoS

(b) NF compositions in the fat-tree topology.

Figure 16: Fat-tree topology and corresponding NF compositions.

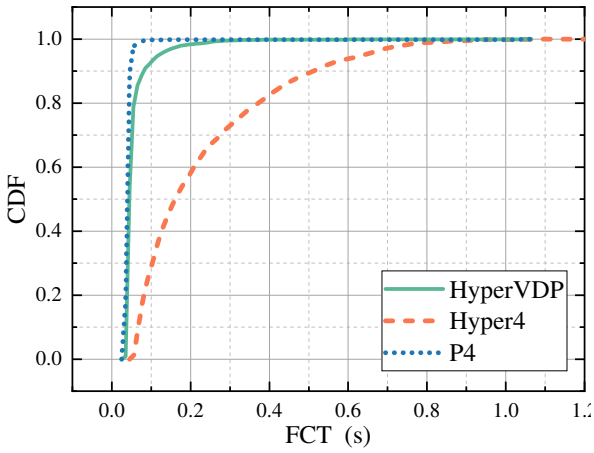
will adopt the composition of *MAC Learn*, *L2_SW*, *IP_SG* and *QoS* at *E3* in another virtual PDP context. These two virtual PDPs can run simultaneously with isolation, and can be dynamically reconfigured with any desired NF compositions without interrupting the physical data plane.

In the fat-tree test bed, every packet generator sends forty elephant flows and one thousand mice flows simultaneously to the randomly-elected peers. The elephant flows contain 100MB data, while the mice flows only contain 10KB data. All the flows are generated by iPerf. In the experiments, we record and analyze the flow-level performance, e.g. the flow completion time (FCT) of mice flows and the throughput of elephant flows.

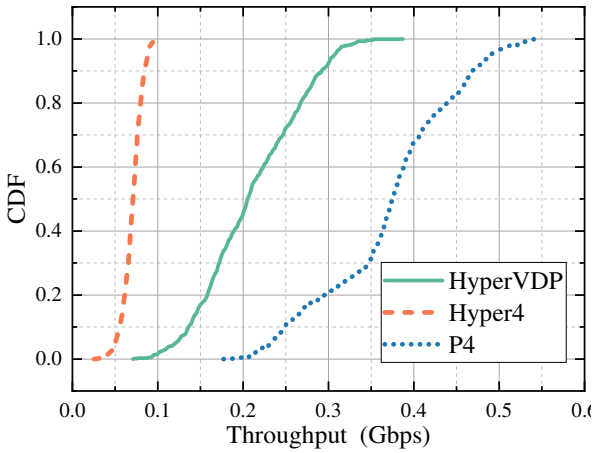
FCT of mice flows on three platforms is shown in Figure 17(a). For Hyper4, the FCT of 90% mice flows is below 0.515 of a second. However, for HyperVDP, 90% of the mice flows can complete within 0.085 of a second, while only 21.9% of the mice flows on Hyper4 can complete within 0.085 of a second. The FCT of mice flows on HyperVDP is much closer to the native P4 who has the optimal performance as the baseline.

Throughput of elephant flows is shown in Figure 17(b). Hyper4 incurs the worst performance among three platforms. In Hyper4, throughput of 90% elephant flows is lower than 85 Mbps. However, for HyperVDP, more than 90% of the flows have the throughput that is larger than 249 Mbps, and more than 90% flows have a throughput up to 293 Mbps. On the native P4, 90% of flows can have throughput of more than 249 Mbps.

The above experiments show that HyperVDP incurs a much smaller performance overhead than Hyper4, which makes HyperVDP a much more well-designed and high-performance



(a) FCT of mice flows.



(b) Throughput of elephant flows.

Figure 17: Cumulative distribution functions for mice flows' FCT and elephant flows' throughput.

hypervisor to introduce virtualization to the PDP. Besides, as we stated above, HyperVDP provisions a stronger ability to support full virtualization of programmable data plane primitives than Hyper4. In a nutshell, HyperVDP, as a hypervisor for the PDP, largely surpasses Hyper4 in the performance dimension and the virtualization capability dimension.

VI. Discussion

Finally, we intend to discuss some open issues of HyperVDP as following:

Relation with NFV and OpenFlow-based virtualization techniques. As mentioned in Section I, HyperVDP aims to provide virtualization of the PDP behavior, which shares a different motivation with NFV and OpenFlow-based virtualization techniques.

As for NFV, HyperVDP is a complementary technique that can be utilized to provide both built-in virtualization in the network as well as performance enhancement. For example, when implementing network function service chain in NFV scenarios, HyperVDP can offload some data-plane-suitable functions and maintain multiple virtual PDPs with different compositions of network functions for different tenant. This

way of hardware and software co-orchestration by NFV and HyperVDP can both enhance the performance and provides service isolation.

Most OpenFlow-based virtualization techniques aim to provide external virtualization which logically merges multiple network nodes into one virtual node. And this motivation is usually achieved by using OpenFlow and other overlay protocols to implement customized packet forwarding behaviors. Unlike OpenFlow-based techniques which use customized forwarding rules to achieve network virtualization, HyperVDP, although motivates differently, can be viewed as an enhanced infrastructure that supports both virtualization of data plane behaviors and customized forwarding rules. For the PDP, forwarding behaviors are decided by both data plane program and forwarding rules. Therefore, comparing with OpenFlow-based techniques, HyperVDP can provide more well-enhanced and customized virtualization functions on the PDP

Capability. The data plane hypervisor needs more research efforts to study the exact capability of the hypervisor to enable offloading NFs onto the device. As a matter of fact, not all NFs are fit for being offloaded onto the PDP and emulated by the hypervisor. For example complex and mutable NFs such as deep packet inspection and sophisticated packet encryption are far from being implemented by the hypervisor. Therefore, the capability of the hypervisor can provide an exact boundary between offloadable NFs and un-offloadable NFs. In this paper, HyperVDP is devoted to facilitating the offloading of those NFs that have enough affinity to the PDP, and is not about which kinds of NFs are suitable for offloading. In the future, we will explore more into the capability of hypervisor on the PDP.

Hardware feasibility. Feasibility of HyperVDP is another important consideration. As described in the paper, the design of HyperVDP is strictly conformed to P4 specification and can be correctly implemented in BMv2 environment. Thus HyperVDP is feasible for P4-enabled hardware devices syntactically. As for hardware deployment restriction, Hyper4 has discussed its deployment feasibility on RMT which is a hardware architecture, and indicates that RMT can support a limited number of applications emulated by Hyper4. As evaluated in Section V, HyperVDP consumes much fewer resources in comparison with Hyper4, which means that RMT can accommodate more applications when being deployed with HyperVDP.

VII. Conclusion

In this paper, we proposed HyperVDP, a high performance hypervisor for full virtualization of the PDP, to provision features of virtualization including the non-exclusive data plane abstraction as well as uninterrupted data plane reconfigurability. In HyperVDP, we propose several innovative techniques such as abstraction of virtual PDP, rapid header parsing, control flow sequencing and dynamic stage mapping, to implement the design on the PDP base the P4 specification. We build the prototype of HyperVDP on BMv2 target and DPDK target. We evaluate the performance of HyperVDP in terms of several benchmarks by comparing HyperVDP with various counterparts. Our evaluation shows that BMv2-target HyperVDP prevails over Hyper4 with 2.5x performance while reducing 4x resource usage. Our DPDK-target HyperVDP can process 64-byte packets at the speed of 42.14 Mpps with a minor throughput penalty, comparing with Open vSwitch at 45.71 Mpps and PISCES at 45.95 Mpps. Meanwhile, the DPDK-target HyperVDP can keep the latency lower than 9 μs with packets of arbitrary size, and forward above 90% of 64-byte packets in less than 7 μs.

Currently, HyperVDP allocates data plane resources in a first-come-first-serve way, and does not detect the inconsistency among network functions. In our future work, we plan to add more features of QoS management into HyperVDP to meet different NFs QoS requirements. Besides,it is non-negligible that virtualization comes with a performance penalty, but it is well worthy to introduce the promising virtualization features into the PDP with an acceptable performance penalty. As more and more NFs could be offloaded into the data plane in the future time, data plane virtualization can be an effective accelerating technique to make the programmable network more dynamic and adaptive.

References

[1] C. Zhang, J. Bi, Y. Zhou, A. Basit, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *2017 26th International Conference on Computer Communication and Networks (ICCCN)*, July 2017, pp. 1–9.

[2] C. Zhang and et. al, "Hyper project," Website, https://github.com/HyperVDP accessed May 2018.

[3] 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 *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, ser. SIGCOMM '13. New York, NY, USA: ACM, 2013, pp. 99–110. [Online]. Available: http://doi.acm.org/10.1145/2486001.2486011

[4] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S.-T. Chuang, I. Keslassy, A. Orda, and T. Edsall, "drmt: Disaggregated programmable switching," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098823

[5] 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. [Online]. Available: http://doi.acm.org/10.1145/2656877.2656890

[6] B. Networks., "Barefoot tofino," Website, https://barefootnetworks.com/technology/#tofino.

[7] Netronome, "Netronome flow processor," Website, https://netronome.com/product/nfp-6xxx/.

[8] B. Company., "High-capacity strataxgs trident 3 ethernet switch series," Website, https://www.broadcom.com/products/ethernet-connectivity/switch-fabric/bcm56870.

[9] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM '17. New York, NY, USA: ACM, 2017, pp. 15–28. [Online]. Available: http://doi.acm.org/10.1145/3098822.3098824

[10] X. Jin, X. Li, H. Zhang, R. Soule, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles*, ser. SOSP '17, 2017.

[11] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. K. Ports, "Just say no to paxos overhead: Replacing consensus with network ordering," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016,

pp. 467–483. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/li

[12] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of tcp," *CoRR*, vol. abs/1611.01529, 2016. [Online]. Available: http://arxiv.org/abs/1611.01529

[13] V. Sivaraman and et al., "Heavy-hitter detection entirely in the data plane," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: ACM, 2017, pp. 164–176. [Online]. Available: http://doi.acm.org/10.1145/3050220.3063772

[14] S. Pack and S. Jang., "Development of universal programmable gateway using p4 and smartnic," Website, https://open-nfp.org/projects/.

[15] N. K. Sharma, A. Kaufmann, T. 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 Implementation (NSDI 17)*. Boston, MA: USENIX Association, 2017, pp. 67–82. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/sharma

[16] Y. Li, R. Miao, C. Kim, and M. Yu, "Flowradar: A better netflow for data centers," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 311–324. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/li-yuliang

[17] B. Li, K. Tan, L. L. Luo, Y. Peng, R. Luo, N. Xu, Y. Xiong, P. Cheng, and E. Chen, "Clicknp: Highly flexible and high performance network processing with reconfigurable hardware," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: ACM, 2016, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934897

[18] Wiki, "Network virtualization," Website, https://en.wikipedia.org/wiki/Network_virtualization.

[19] M. Casado and N. McKeown, "The virtual network system," in *Proceedings of the 36th SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '05. New York, NY, USA: ACM, 2005, pp. 76–80. [Online]. Available: http://doi.acm.org/10.1145/1047344.1047383

[20] ETSI, "Network functions virtualisation," Website, http://www.etsi.org/technologies-clusters/technologies/nfv.

[21] V. Corporation, "Vmware and virtual machine," Website, https://www.vmware.com/cn.html.

[22] VMware., "vsphere esxi bare-metal hypervisor," Website, http://www.vmware.com/products/esxi-and-esx.html.

[23] D. Hancock and J. V. D. Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," ser. CoNEXT '16, 2016, pp. 35–49.

[24] Y. Liao, D. Yin, and L. Gao, "Pdp: Parallelizing data plane in virtual network substrate," in *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*, ser. VISA '09. New York, NY, USA: ACM, 2009, pp. 9–18. [Online]. Available: http://doi.acm.org/10.1145/1592648.1592651

[25] M. B. Anwer and N. Feamster, "Building a fast, virtualized data plane with programmable hardware," *SIGCOMM Comput. Commun. Rev.*, vol. 40, no. 1, pp. 75–82, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1672308.1672323

[26] J. Liu and et al, "Building a flexible and scalable virtual hardware data plane," in *Proceedings of the 11th International IFIP TC 6 Conference on Networking - Volume Part I*, ser. IFIP'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 205–216. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-30045-5_16

[27] E. Kohler, "The click modular router," Ph.D. dissertation, Cambridge, MA, USA, 2001, aAI0803026.

[28] T. Taleb, M. Bagaa, and A. Ksentini, "User mobility-aware virtual network function placement for virtual 5g network infrastructure," in *2015 IEEE International Conference on Communications (ICC)*, June 2015, pp. 3879–3884.

[29] I. Benkacem, T. Taleb, M. Bagaa, and H. Flinck, "Optimal vnfs placement in cdn slicing over multi-cloud environment," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 616–627, March 2018.

[30] A. Laghrissi, T. Taleb, M. Bagaa, and H. Flinck, "Towards edge slicing: Vnf placement algorithms for a dynamic amp;amp; realistic edge cloud environment," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, Dec 2017, pp. 1–6.

[31] M. Bagaa, T. Taleb, and A. Ksentini, "Service-aware network function placement for efficient traffic handling in carrier cloud," in *2014 IEEE Wireless Communications and Networking Conference (WCNC)*, April 2014, pp. 2402–2407.

[32] M. B. D. D. R. A. Addad, T. Taleb and H. Flinck, "Towards modeling cross-domain network slices for 5g," in *IEEE Globecom18*, December 2018.

[33] M. Bagaa, T. Taleb, A. Laghrissi, and A. Ksentini, "Efficient virtual evolved packet core deployment across multiple cloud domains," in *2018 IEEE Wireless Communications and Networking Conference (WCNC)*, April 2018, pp. 1–6.

[34] P4 Language Consortium., "P4-hlir," Website, https://github.com/p4lang/p4-hlir.

[35] ——, "P4-bmv2," Website, https://github.com/p4lang/behavioral-model.

[36] M. Budiu, "Compiling p4 to ebpf," Website, https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4.

[37] P4 Language Consortium., "A control plane framework and tools for the p4 programming language," Website, https://github.com/p4lang/PI.

[38] T. open networking operating system (ONOS)., "P4 support via bmv2 and p4runtime," Website, https://wiki.onosproject.org/display/ONOS/P4+support+via+BMv2+and+P4Runtime.

[39] R. Harrison, Q. Cai, A. Gupta, and J. Rexford, "Network-wide heavy hitter detection with commodity switches," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '18. New York, NY, USA: ACM, 2018, pp. 8:1–8:7. [Online]. Available: http://doi.acm.org/10.1145/3185467.3185476

[40] A. Schwabe and H. Karl, "Using mac addresses as efficient routing labels in data centers," in *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, ser. HotSDN '14. New York, NY, USA: ACM, 2014, pp. 115–120. [Online]. Available: http://doi.acm.org/10.1145/2620728.2620730

[41] P. D. A and H. J. L, "Computer organization and design: the hardware software interface," 1998.

[42] Intel, "Intel dpdk," Website, http://www.dpdk.org/.

[43] B. Pfaff, J. Pettit, T. Koponen, E. J. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'15. Berkeley, CA, USA: USENIX Association, 2015, pp. 117–130. [Online]. Available: http://dl.acm.org/citation.cfm?id=2789770.2789779

[44] 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: ACM, 2016, pp. 525–538. [Online]. Available: http://doi.acm.org/10.1145/2934872.2934886

[45] I. Corporation., "Intel 82599 10 gigabit ethernet controller: Datasheet." Website, https://www.intel.com/content/www/us/en/embedded/products/networking/82599-10-gbe-controller-datasheet.html.

[46] iPerf Project., "iperf - the ultimate speed test tool for tcp, udp and sctp." Website, https://iperf.fr/.

[47] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of the 2015 Internet Measurement Conference*, ser. IMC '15. New York, NY, USA: ACM, 2015, pp. 275–287. [Online]. Available: http://doi.acm.org/10.1145/2815675.2815692