# HyperV: A High Performance Hypervisor for Virtualization of the Programmable Data Plane

Cheng Zhang*, Jun Bi*, Yu Zhou*, Abdul Basit Dogar*, Jianping Wu*
*Tsinghua University
zhang-cheng13@mails.tsinghua.edu.cn, junbi@tsinghua.edu.cn,
{y-zhou16, bas15}@mails.tsinghua.edu.cn, jianpingwu@tsinghua.edu.cn

***Abstract*—P4 is a domain specific language designed to define behaviors of the programmable data plane. It facilitates offloading hardware-suitable Network Functions (NFs) to a data plane. Consequently, NFs can maximally benefit from high performance of hardware devices, meanwhile more CPU power can be reserved for user applications. However, since the programmable data plane provides an NF with an exclusive network context, different NFs cannot operate on the same data plane simultaneously. Besides, it is hardly possible to dynamically reconfigure programmable network devices without interrupting the operation of a data plane.**

**Therefore, we propose HyperV, a high performance hypervisor for virtualization of a P4 specific data plane, to provide both non-exclusive and uninterrupted features.We implemented HyperV based on a P4-BMv2 target and a DPDK target respectively. Then we evaluated BMv2-target HyperV by comparing with Hyper4, a recently proposed hypervisor, and evaluated DPDK-target HyperV by comparing with PISCES and Open vSwitch. Results show that BMv2-target HyperV averagely prevails over Hyper4 2.5x in performance while reducing resource usage by 4x. DPDK-target HyperV performs comparably to Open vSwitch and PISCES, with the worst case of a throughput penalty in less than 7%, while providing a powerful capability of virtualization which neither of them provides.**

## I. Introduction

Software Defined Networking (SDN) has started an era providing network operators with a programmable control over their networks. P4 [1] and POF [2] are Domain Specific Languages (DSLs) that intend to define the behavior of a data plane. With the help of DSLs and programmable network devices, in Network Function Virtualization (NFV) scenarios, data plane affiliative Network Functions (NFs) can be offloaded as data plane programs, e.g. offloading a TCP monitor [3], to save CPU power for user applications and benefit from high performance and parallel acceleration of network devices. Most recent researches and industrial cases, including ClickNP [4] and [5], [6] etc., also confirm this offloading technique can significantly improve performance of networks. With the foreseeable flourish of DSLs and programmable network devices in the future, more and more suitable NFs could be considered shifting onto data planes for purpose of improving network performance and device efficiency.

However, several barriers from programmable network devices hinder this promising offloading technique. Each data plane provided by a P4-capable device represents a view of the physical data plane, which means a data plane is exclusively operated by a program once configured. And this leads to the result that various programs cannot run simultaneously and share underlying resources in a non-exclusive way on a single data plane. However, in many NFV scenarios, with the aim of supporting multi-tenancy and improving device efficiency, it is necessary to run diverse programs on a given programmable network device.

Additionally, due to intrinsic constraints of programmable network devices, it is hardly possible to dynamically reconfigure a network device without interrupting the operation of a data plane. According to [4] and [7], reconfiguration of a programmable network device usually depends on the size and design of the chip ranging from tens of milliseconds to several minutes. Besides, device states and table entries will get lost due to this interruption. Therefore, in order to maintain the consistency of network states, it is exceedingly necessary to provide operators with an uninterrupted way to dynamically load programs onto data planes.

Virtualization of a programmable data plane (PDP) is one of many possible solutions to these problems. By virtualizing a PDP, different virtual PDPs can be created on one single physical PDP, which has the similar concept of virtual machines in computer virtualization. In this way, programs can be dynamically instantiated/migrated/deleted at runtime on virtual PDPs without interrupting the physical PDP, and different virtual PDPs can share the same physical PDP simultaneously. Additionally, in order to implement virtualization, a hypervisor of PDPis designed to interpret programs and manage virtual PDPs on top of a physical PDP.

A recent research, Hyper4 [8], proposed a potential way of virtualizing a PDP. In Hyper4, a hypervisor-like program is designed to provide partial virtualization of P4 data plane, so that certain programs can be equivalently emulated in Hyper4 to achieve non-exclusive and uninterrupted features. However, Hyper4 has several problems as a data plane hypervisor. (i) Hyper4 lacks a structural design and well-abstracted models to interpret P4 programs. (ii) Hyper4 is partial virtualization of P4 specific data plane, and several essential P4 language elements such as *if-else* statement with boolean expression, control flow, are absent. Therefore, this defect make Hyper4 not adaptable for most P4 programs. Additionally, (iii) Hyper4 suffers from a severe performance penalty due to the heavy use of resubmitting actions in the parser and complex inner control logic. Lastly, (iv) Hyper4 also suffers from an excessive usage of hardware resources due to its hard-coded implementation, which makes it not applicable to programs with an arbitrary number of stages.

Inspired by Hyper4, we make a step further and propose HyperV, a hypervisor with a structural design and well-abstracted models, to fully virtualize a P4 data plane while providing high performance and efficient use of hardware resources. In HyperV, we proposed several novel techniques, including control flow sequencing, dynamic stage mapping, to achieve full virtualization of P4 data plane. Therefore, arbitrary P4 programs can be instantiated on a virtual PDP transparently. 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 dramatically improve efficiency. Moreover, techniques of rapid parsing and pipeline bypassing are used to reduce the performance penalty. Hence, HyperV can greatly attain the balance between virtualization and performance. Apart from the models mentioned above, HyperV also provides operators with a set of operating models, such as a program, a stage, to support a flexible way of composing NFs.

We have built two proof-of-concept prototypes of HyperV based on the P4-BMv2 target and the DPDK target respectively. We compared BMv2-target HyperV with Hyper4. Results show that BMv2-target HyperV is averagely 2.5x performance advance of Hyper4 in terms of bandwidth and latency while reducing 4x resource usage. Then we evaluated DPDK-target HyperV by comparing with state-of-the-art software switches, PISCES and Open vSwitch. DPDK-target HyperV 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, HyperV can maintain the line rate of 40 Gbps. Meanwhile, DPDK-target HyperV can keep the forwarding latency lower than 9 μs with packets of arbitrary size, and above 90% of 64-byte packets can pass through HyperV in less than 7 μs.

In this paper, we make the following contributions:

- We designed HyperV as a hypervisor that fully virtualizes the P4 data plane while maintaining high performance and resource efficiency.
- We proposed an architecture of a programmable data plane hypervisor and a virtual PDP to represent the virtual view of a physical PDP.
- We proposed several novel techniques and creative models to facilitate virtualization of a PDP.
- We implemented a BMv2-target and a DPDK-target HyperV respectively, and evaluated them by comparing with their counterparts. Results show that HyperV can achieve virtualization with a minor performance penalty.

We will discuss the related work in next section, then show some P4 language basics in Section III. In Section IV, we will illustrate the design of HyperV and some key techniques as well as models. In Section V, we will evaluate BMv2-target HyperV and DPDK-target HyperV respectively. After that we will discuss capability and hardware feasibility of HyperV in Section VI, and make a conclusion in Section VII.

## II. Related Work

To the best of our knowledge, Hyper4, firstly, proposed a hypervisor-like program on a P4 specific PDP, and is the only research that shares the same motivation with HyperV in providing virtualization features. In spite of problems we mentioned in Section I, Hyper4 does show the potential to introduce a hypervisor onto the programmable data plane and the useful features provided by virtualization.

Other researches related with offloading, e.g. [9], [4], [10], [11], 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 [12], 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.

Authors in [10] 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 [10] is confined to a fixed router function and the reconfiguration of a new NF will interrupt the data plane. Based on the FPGA hardware device, [11] 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 [11] is that HyperV is devoted to designing a systematic approach to virtualize the protocol-independent and programmable data plane. And [11] mainly focuses on slicing the OpenFlow-like data plane to afford an isolation feature.

## III. Background of P4 Language Basics

P4, Programming Protocol Independent Packet Processor, is a recently proposed high level language for defining the behavior of a data plane. A P4 program mainly defines (i) a parser, containing a collection of protocol formats and corresponding state machines to parse headers; (ii) a Direct Acyclic Graph (DAG) of match-action stages, named control flow, to define how packets are processed; and (iii) 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 [13]. Then a back end compiler, e.g. p4c-bmv2 [14] or P4-to-EBPF [15], will adapt the program into different targets including FPGA, CPU, etc. Finally, through a runtime controller, operators can

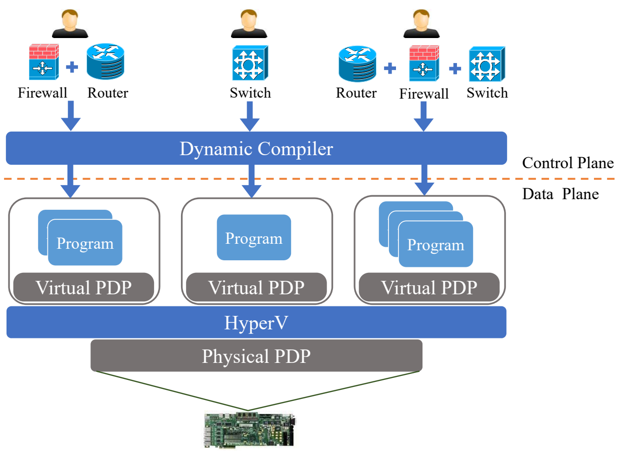


Figure 1. HyperV design overview.

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.

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

## IV. Design of HyperV

### A. Overview

Figure 1 shows the overall architecture of HyperV. HyperV acts as a hypervisor on top of a physical P4 specific data plane and provides a virtual view of underlying hardware resources to upper programs. Operators can dynamically configure HyperV 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 HyperV from the control plane, we also design a dynamic compiler running in an SDN controller. It can compile and analyze P4 programs created by operators, then transform P4 language elements into data plane models managed by HyperV, and dynamically allocate resources such as program id, stage id, stage slot, and populate tables on the virtual PDP.

In HyperV, we use several novel techniques and models to achieve high performance as well as full virtualization of P4 language elements. In the following section, we will provide details of several key models and techniques used in HyperV.

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

### B. Rapid Parsing

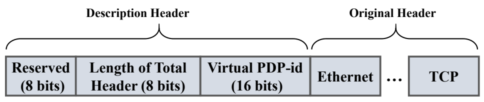


Figure 2. Example of description header.

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 pattern 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 any resubmitting action.

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. [16]). 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

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

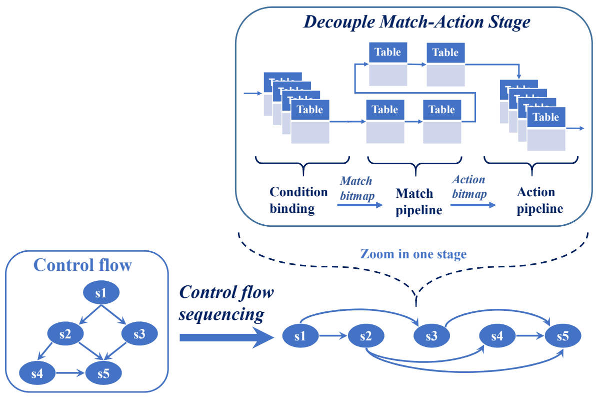


Figure 3. Control flow sequencing.

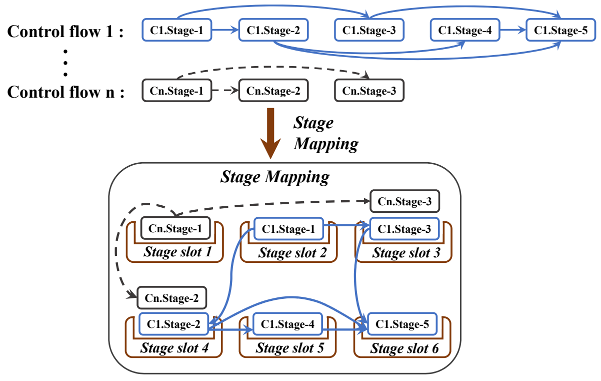


Figure 4. Dynamic stage mapping.

various DAGs into a unified pattern. Hence, as shown in Figure 3, firstly (i) 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 HyperV 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, HyperV can support a compound action with an arbitrary number of primitives.

After decoupling a match-action stage, secondly (ii) 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

After control flow sequencing, HyperV 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, HyperV faces the challenge of mapping an unpredictable sequence of stages onto the hardware data plane with limited resources. Inspired by the idea of virtual memory mapping in the operating system, we designed the dynamic stage mapping as shown in Figure 4.

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, HyperV 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.

Based on the stage mapping technique, HyperV not only supports the stage sequence with arbitrary length, but also provides operators with an infrastructural model of the stage slot. Through 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

Table I
TABLE DECLARATION IN HYPERV.

| | Config | other | Stage slot | | |
|---|---|---|---|---|---|
| | | | Condition binding | Match pipeline | Action pipeline |
| Number of tables | 2 | 3 | 4 | 4 | 16 |

the program, instantiates the virtual PDP as well as other necessary states including program id, stage id etc., and finally populates table entries. In HyperV, each packet is 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.

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 AND ANALYSIS OF HYPERV

At the time of writing this paper, we could not manage the experiment on a P4 capable hardware device [7]. Therefore, we implemented HyperV in the standard P4 BMv2 [14] environment, and compared BMv2-target HyperV with Hyper4 in terms of resource usage and performance. In order to further exploit the feasibility and performance penalty of HyperV, we also implemented HyperV on the Intel DPDK [17] platform and evaluated DPDK-target HyperV by comparing with Open vSwitch [18] and PISCES [19], which are state-of-the-art software switches, in terms of several benchmarks.

Our BMv2-target HyperV has a line-of-code (LoC) of 2000, while DPDK-target HyperV has a LoC of 8000. In the following section, we will elaborate the evaluation of BMv2-target HyperV and DPDK-target HyperV separately.

### A. Evaluation of BMv2-target HyperV

All of our tests for BMv2-target HyperV are executed on the off-the-shelf x86 platform with 2×4 Intel E5-2637 3.50Ghz cores and 64GB memory. For resource usage, we evaluate HyperV in terms of three significant benchmarks. First one is resource declaration of a hypervisor. This benchmark denotes how many resources are declared in the hypervisor. Since hardware devices have their resource limitations, a hypervisor should always declare resources as less as possible to be more efficient and deployable. Second is table usage in runtime. This benchmark identifies how many tables are used when emulating an NF by a hypervisor. It directly impacts the latency of packets. Third is metadata declaration. It involves

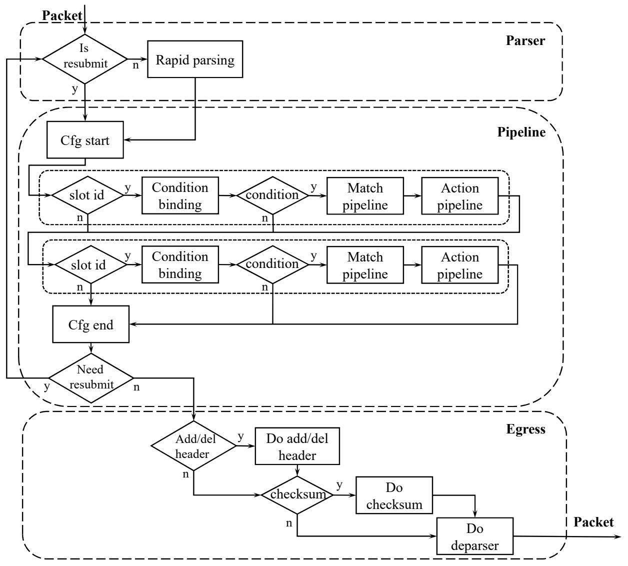


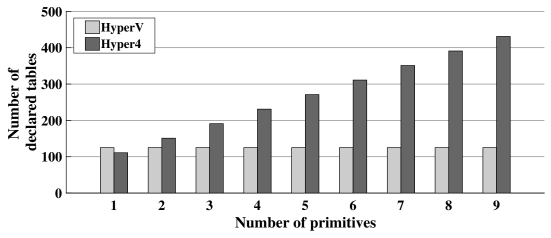
Figure 5. Example of HyperV with two slots.

the width and type of match fields in a table, and has an impact on TCAM and energy. For performance, we evaluate BMv2-target HyperV by comparing with Hyper4 running different programs we have implemented in terms of latency and bandwidth.

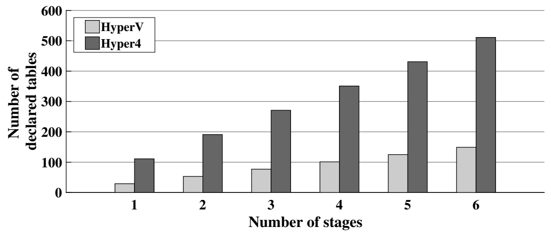
*1)* ***Table declaration in HyperV:*** Figure 5 shows the processing flow of HyperV. There are two configuration tables, one is for setting the address of first stage for each packet at the beginning; the other is for resubmitting the packet back into the ingress pipeline in case of insufficient slots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

*2)* ***Comparison of table declaration:*** Hyper4 hard-codes the implementation of a stage pipeline, hence the number of table declaration is closely related to the number of stages hard-coded as well as the maximum number of primitives permitted in one compound action. However, table declaration in HyperV 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 HyperV.

In Figure 6(a), Hyper4 and HyperV both hard-code 5 stages (5 slots in HyperV). As the maximum number of primitives permitted in one compound action increases, Hyper4 suffers from a linear growth of table declaration, while HyperV still remains the same. In Figure 6(b), both Hyper4 and HyperV set the maximum number of primitives to 9. We can see that even in the case of 6 slots, HyperV declares only 149 tables while Hyper4 needs more than 500 tables.



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



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

Figure 6. Table declaration in different cases.

Table II
METADATA DECLARATION.

| Platform | Metadata(Bits) | | | |
|---|---|---|---|---|
| | Header | User | Control | Total |
| HyperV | 800 | 256 | 312 | 1368 |
| Hyper4 | 800 | 256 | 2256 | 3312 |

*3) **Comparison of metadata declaration**:* Table II shows metadata declaration in Hyper4 and HyperV. It can be seen that because a hypervisor cannot infer the length of a packet header until it starts to process the packet. Hence, both Hyper4 and HyperV reserve 800 bits (far surpasses common packet header length) to hold a packet header, and 256 bits for user-defined metadata (usually enough for most programs). Benefiting from a rather structural design and simple inner control logic, HyperV only needs 312 bits of metadata for intrinsic control, while Hyper4 needs as many as 2256 bits.

*4) **Comparison of table usage in runtime** :* Table III shows the number of tables used in runtime for various programs. In our experiment, we implemented four programs and tested them independently. A *L2 switch* is a simple layer-2 switch with the learning ability. A *Firewall* is a layer-3 and layer-4 firewall which can process packets based on user rules. A *Router* can process packets with the longest prefix match. An *ARP proxy* simply processes ARP request and reply packets.

As *L2 switch* is shown, packets in a native P4 switch flow through two match-action tables, one is a source MAC table and another is a destination MAC table. Accordingly in Hyper4, 13 tables are required to emulate the *L2 switch*. While in HyperV, only 5 tables are enough, including one configuration table at the beginning, two match tables and two action tables. The *L2 switch* in HyperV 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 in runtime.

Table III
TABLE USAGE IN RUNTIME FOR DIFFERENT PROGRAMS.

| Programs | Native P4 | Hyper4 | HyperV |
|---|---|---|---|
| L2 switch | 2 | 13 | 5 |
| Firewall | 3 | 22 | 8 |
| Router | 4 | 28 | 16 |
| ARP proxy | 4 | 48 | 10 |

Table IV
LATENCY RATIO (PING TEST).

| Programs | Hyper4 : Native P4 | HyperV : Native P4 |
|---|---|---|
| L2 switch | 3.41 : 1 | 1.70 : 1 |
| Firewall | 4.71 : 1 | 2.09 : 1 |
| L2 switch + Firewall | 3.44 : 1 | 2.23 : 1 |

Table V
BANDWIDTH RATIO (IPERF3 TEST).

| Programs | Hyper4 : Native P4 | HyperV : Native P4 |
|---|---|---|
| L2 switch | 0.17 : 1 | 0.49 : 1 |
| Firewall | 0.11 : 1 | 0.43 : 1 |
| L2 switch + Firewall | 0.17 : 1 | 0.35 : 1 |

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

*5) **Comparison of performance**:* Since we could not manage to port Hyper4 into our environment. In order to make a horizontal comparison with Hyper4, we port all programs provided in the source code of Hyper4 into our environment without any modification. Then test different compositions of programs in accordance with Hyper4 and calculate the ratio of Hyper4 and HyperV to native P4 in each own environment. Table IV and Table V show the results in terms of latency and bandwidth respectively.

As for HyperV, we can see the performance decreases when programs become more complicated. As for Hyper4, we suppose the test rules for *L2 switch + Firewall* are rather simple, which makes it almost the same as the *L2 switch*. In respect of latency, HyperV is about 2x of the native switch, while Hyper4 is 3x to 4x of the native switch. As for bandwidth, HyperV averagely improves 2x to 3x of bandwidth performance comparing with Hyper4. We attribute this improvement to a much less use of resources in runtime and a well-designed structure of HyperV.

## B. Evaluation of DPDK-target HyperV

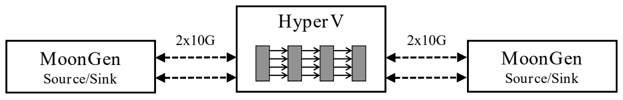


Figure 7. Testbed topology.

*1) **Setup and metrics**:* Figure 7 shows the testbed topology for evaluating the performance of DPDK-target HyperV. Our testbed contains three x86 servers, two of which with 64GB memory and 2×6 Intel(R) Xeon(R) E5-2620 2.40GHz cores, are running MoonGen [20] packet generators and receivers. 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. The software switches and HyperV are running on the middle server which is equipped with 64GB memory and 2×4 Intel(R) Xeon(R) E5-2637 3.50GHz cores. This server has four 10G NIC ports, which are directly connected to the packet generators and receivers through optical fibers. Hence, the testbed could generate customized packets at 40 Gbps to evaluate the forwarding performance of HyperV against Open vSwitch and PISCES.

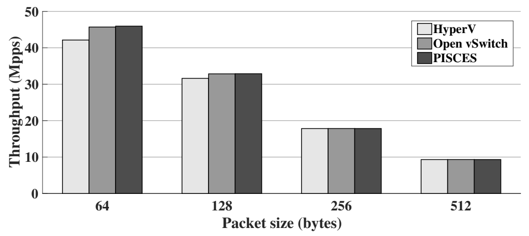
Open vSwitch has multiple data plane drivers, including the kernel driver and the DPDK driver. Due to the fact that HyperV is built on DPDK, we choose the DPDK-based Open vSwitch as a high performance software counterpart in this paper. PISCES [19] is a programmable and protocol-independent software 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 HyperV, PISCES, and Open vSwitch in terms of throughput and latency. All 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 HyperV. We utilized the time-stamp counter library of DPDK, and measured the number of CPU cycles consumed by inner elements of HyperV.

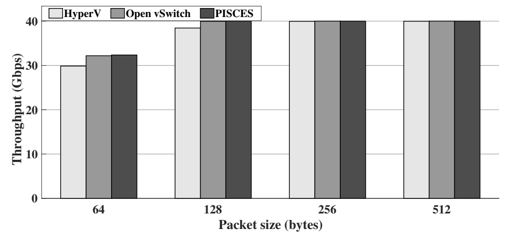
*2) **Macro benchmark**:* The packet generators in Figure 7 can create the specific size of packets at the full speed of NICs. As for forwarding latency, sample packets with time stamps will be sent to measure the end-to-end delay of software switches. MoonGen will pour the background traffic containing 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 is shown in Figure 8(a) and 8(b), which illustrate that the DPDK-target HyperV implementation has a minor performance penalty while introducing virtualization onto a PDP. As shown, HyperV 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. HyperV introduces a throughput penalty of about 7% in terms of processing small packets. For large packets, all the three switches can maintain a line-rate throughput.

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



(a) Forwarding performance in packets per second.



(b) Forwarding performance in Gigabits per second.

Figure 8. Throughput comparison.

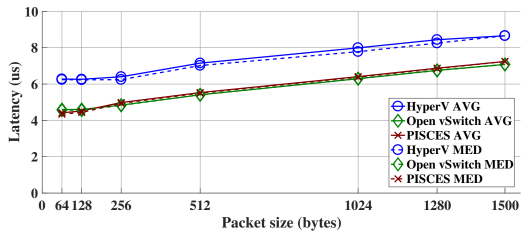
enhancing forwarding performance, while the existing prototype of DPDK-target HyperV does not use any optimization technique of flow caching to reduce the throughput penalty. In spite of the throughput penalty comparing with PISCES and Open vSwitch, we still deem it worthy to introduce virtualization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

**Latency** of packets is another concerned aspect in our benchmarks. The average number (*AVG*) and median number (*MED*) of latency are plotted in Figure 9(a). As it can be seen, the latency of packets increases slightly with the packet size. The latency of packets produced by Open vSwitch and PISCES ranges from 4.42 μs to 7.24 μs. HyperV can process a 64-byte packet in 6.28 μs on average. Besides, Figure 9(b) shows the latency cumulative distribution of 64-byte packets and indicates that more than 93.24% of 64-byte packets can pass through HyperV in no more than 7 μs.

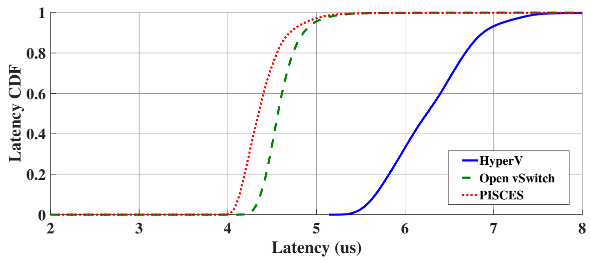
Both of PISCES and Open vSwitch adopt the run-to-completion model as their packet processing model. In the run-to-completion model, each packet in the data path is assigned a single thread that completes the full functionality of packet processing at one time. Nevertheless, DPDK-target HyperV is based 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 going through the pipeline of HyperV is theoretically higher than PISCES and Open vSwitch.

As aforementioned, the goal of this evaluation is to inspect the performance penalty introduced by HyperV accurately. We can infer from the results that the performance penalty of HyperV 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 HyperV in depth, we make further efforts to



(a) Latency of packets with different sizes.



(b) Latency cumulative distribution of 64-byte packet.

Figure 9. Latency comparison.

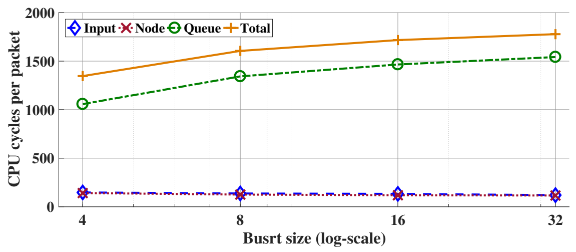


Figure 10. CPU cycle consumption.

measure CPU cycles consumed by inner elements of HyperV. The prototype of HyperV has several processing elements which can repeatedly process bursts of packets. (i) 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. (ii) 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. (iii) The last kind of element is the conditional node which can process a conditional statement. Through this node, HyperV could implement arbitrary conditional semantics of P4 language or other data plane DSLs.

As stated in Section V-B2, the increase of latency is partially introduced by the queue model implemented in DPDK-target HyperV. Moreover, since elements in HyperV are designed to process batches of packets, the size of a batch might also has an impact on latency. Thus, we inspected the influence of a queue on latency 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 HyperV. 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 10. Additionally, we also measured CPU cycles consumed by different selected actions in HyperV. We will illustrate the results separately in the following.

**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 10, 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 a input node and a general node decrease by several cycles with the increase of batch size, because the packet batch could amortize the cost of processing, e.g. memory accesses, function calls, over packets in the batch. Thus, it is reasonable to infer that the latency penalty partially originates from the slightly excessive use of the queue in HyperV. 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 smaller batch size leads to lower latency 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 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 on average. What's more, REMOVE_HEADER only needs about 45 cycles.

## VI. Discussion

Finally, we intend to discuss some open issues on the capability and hardware feasibility of HyperV as following:

**Capability** of a hypervisor needs more research efforts to support offloading network functions. Not all NFs are fit for being offloaded onto the programmable data plane and emulated by a hypervisor. For example complex and mutable network functions such as deep packet inspection and sophisticated packet encryption are far from being implemented by the hypervisor. Actually, HyperV is devoted to facilitating the offloading of NFs that have enough affinity to the data plane. And our emphasis in this paper is not about which kinds of NFs are suitable for offloading onto the data plane but how to offload various kinds of NFs through virtualization of a programmable data plane.

**Hardware feasibility** of HyperV is another important consideration. When this paper is written, we could not manage to experiment on a P4-capable hardware device. However, since our design is strictly conformed to P4 specification and can be correctly implemented in BMv2 environment, HyperV is feasible for P4 hardware devices syntactically. As for deployment restriction, Hyper4 has discussed its deployment on RMT [21] which is a hardware architecture for P4, and indicates that RMT supports a limited number of applications emulated by Hyper4. As stated in Section V-A, HyperV consumes much fewer resources comparing with Hyper4, which means that RMT can accommodate more applications when deploying HyperV on it. Besides, the performance penalty may hinder HyperV from being a practical tool. Thus, in our future work, we will pay more attention on continuous improvement and optimization for HyperV. Moreover, we will step further to make HyperV closer to reality by evaluating it on more hardware platforms.

## VII. Conclusion

In this paper, we proposed HyperV, a high performance hypervisor for full virtualization of P4 specific data plane, to benefit from features of virtualization as well as minimize performance penalty. Our evaluation shows that BMv2-target HyperV prevails over Hyper4 with 2.5x performance while reducing 4x resource usage. Our DPDK-target HyperV 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 HyperV 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.

It is non-negligible that HyperV comes with a performance penalty, but it is well worthy to introduce the promising virtualization features into the programmable data plane with an acceptable performance penalty. Because we reasonably consider virtualization of the data plane as an up-and-coming topic that can effectively accelerate researches of offloading NFs and modeling more general forwarding elements in network devices. As HyperV is under extensive developing and needs further improvement and optimization, we look forward to collaborate with more researchers who share the same belief with us to innovate on virtualization of the programmable data plane.

## VIII. Acknowledgements

We are grateful to the anonymous reviewers for their insightful and constructive comments, as well as our colleagues, Yangyang Wang, Chen Sun and Zhaogeng Li, for their helpful suggestions. This research is sponsored by the National Natural Science Foundation of China (No.61472213 and No.61502267) and National Research Foundation of Korea (NRF-2014K1A1A2064649). Jun Bi is the corresponding author.

## References

[1] Cole Bosshart, Pat ainger and et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95, July 2014.

[2] Haoyu 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*, HotSDN '13, pages 127–132, New York, NY, USA, 2013. ACM.

[3] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. Dapper: Data plane performance diagnosis of tcp. *CoRR*, abs/1611.01529, 2016.

[4] Bojie Li and et al. Clicknp: Highly flexible and high performance network processing with reconfigurable hardware. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 1–14, New York, NY, USA, 2016. ACM.

[5] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. Netfpga: Reusable router architecture for experimental research. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 1–7, New York, NY, USA, 2008. ACM.

[6] Erik Rubow, Rick McGeer, Jeff Mogul, and Amin Vahdat. Chimpp: A click-based programming and simulation environment for reconfigurable networking hardware. In *Proceedings of the 6th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '10, pages 36:1–36:10, New York, NY, USA, 2010. ACM.

[7] Barefoot Networks. Barefoot tofino. Website. https://barefootnetworks.com/technology/#tofino.

[8] David Hancock and Jacobus Van Der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. CoNEXT '16, pages 35–49, 2016.

[9] Yong Liao and et al. Pdp: Parallelizing data plane in virtual network substrate. In *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*, VISA '09, pages 9–18, New York, NY, USA, 2009. ACM.

[10] Muhammad Bilal Anwer and Nick Feamster. Building a fast, virtualized data plane with programmable hardware. *SIGCOMM Comput. Commun. Rev.*, 40(1):75–82, January 2010.

[11] Junjie 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*, IFIP'12, pages 205–216, Berlin, Heidelberg, 2012. Springer-Verlag.

[12] Eddie Kohler. *The Click Modular Router*. PhD thesis, Cambridge, MA, USA, 2001. AAI0803026.

[13] Barefoot Networks. P4-hlir. Website. https://github.com/p4lang/p4-hlir.

[14] Barefoot Networks. P4-bmv2. Website. https://github.com/p4lang/behavioral-model.

[15] Mihai. Budiu. Compiling p4 to ebpf. Website. https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4.

[16] Arne Schwabe and Holger Karl. Using mac addresses as efficient routing labels in data centers. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 115–120, New York, NY, USA, 2014. ACM.

[17] Intel. Intel dpdk. Website. http://www.dpdk.org/.

[18] Ben Pfaff and et al. The design and implementation of open vswitch. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 117–130, Berkeley, CA, USA, 2015. USENIX Association.

[19] Muhammad et al Shahbaz. Pisces: A programmable, protocol-independent software switch. In *Proceedings of the 2016 Conference on ACM SIGCOMM 2016 Conference*, SIGCOMM '16, pages 525–538, New York, NY, USA, 2016. ACM.

[20] Paul Emmerich and et al. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, IMC '15, pages 275–287, New York, NY, USA, 2015. ACM.

[21] Pat Bosshart and et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 99–110, New York, NY, USA, 2013. ACM.