This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/OJCOMS.2020.2990182, IEEE Open Journal of the Communications Society

Virtualization in Programmable Data Plane: A Survey and Open Challenges

Sol Han, Seokwon Jang, Hongrok Choi, Hochan Lee, and Sangheon Pack

Programmable data plane (PDP) is an emerging technology for programming packet processing tasks by means of a domain-specific high-level language (e.g., programming protocol-independent packet processor (P4)) and programmable switch chips. Recently, several PDP virtualization schemes have been introduced to enable more flexible and elastic network management. In this article, we first give an overview PDP and P4. After that, existing PDP virtualization schemes are classified into hypervisor- and compiler-based approaches and their pros and cons are analyzed in detail. Finally, open challenges for PDP virtualization are identified and future research directions are presented.

Index Terms—**Programmable data plane, P4, Virtualization.**

I. Introduction

With the advancements in programmable switch chips and data plane programming languages, programmable data plane (PDP) is perceived as an emerging technology. The key feature of PDP is the programming of packet processing operations by means of high-level and domain-specific languages. Beyond supporting the fixed functions in the existing switch chip, the aforementioned PDP feature gives network operators or developers the flexibility to decide packet processing tasks in the way they want. Such flexible data plane enables rapid deployment of new data plane functions and their prototyping. In addition, there is no need to re-design the application-specific integrated circuits (ASICs) of switches to implement new data plane functions, which results in a significant saving of capital expenditure (CAPEX). Moreover, the performance of programmable switch chips has evolved to such an extent that it nearly meets the performance of the existing fixed switch chips. These advantages make PDP an alternative and promising technology for the next-generation data center networks.

One of the most active studies on PDP is related to the offloading of network functions (NFs) (e.g., load balancer [12] and network cache [13]) that conventionally operate on the server. These studies assume a network function virtualization (NFV) environment, which inevitably requires multi-tenancy for practical scenarios. On the other hand, there have been attempts to build a global network testbed using PDP for testing emerging technologies, which should support multiple users' applications or programs to operate on a shared data plane [1], [2]. However, current PDP does not support any multi-tenancy (i.e., virtualization) and therefore it is difficult to run multiple NFs simultaneously in an isolated manner on a single programmable switch.

To address this issue, PDP virtualization needs to be supported, which allows multiple logical (or virtual) data planes on a physical data plane. Although some notable methods have been proposed for PDP virtualization in the literature, they have both advantages and disadvantages, and several issues are still open. Therefore, we survey the state-of-the-art on the PDP virtualization and identify open challenges. We first provide an overview of the fundamentals of PDP, i.e., protocol independent switch architecture (PISA) [17] and programming protocol-independent packet processor (P4) [4]. After that, the literature on PDP virtualization is classified into 1) hypervisor-based approach and 2) compiler-based approach, and their advantages and disadvantages are analyzed in detail. Through qualitative and quantitative analyses, the open challenges and their possible solutions are discussed.

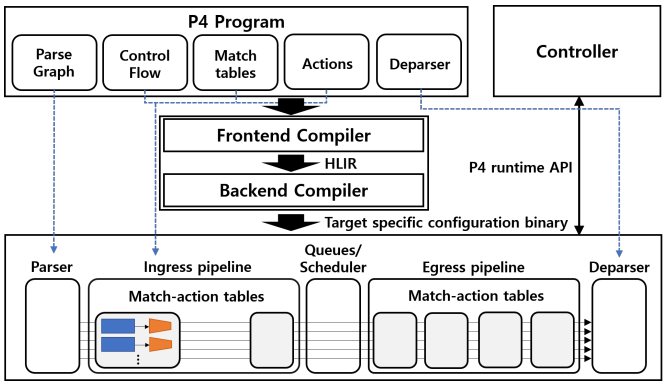

Fig. 1: Workflow of data plane programming.

II. Overview of PDP

In this section, we briefly introduce a well-known programmable switch architecture (i.e., PISA) and a representative programming language (i.e., P4).

A. PISA

PISA is a programmable switch model that generalizes re-configurable match-action tables (RMT) [11]. As shown in Figure 1, PISA consists of 1) a parser, 2) ingress/egress pipelines, and 3) a deparser. The programmable parser extracts headers from the packet arriving at the programmable switch. The extracted headers are then sent to the pipeline stages to

This work was supported in part by IITP grant funded by the Korea government (MSIT) (No. 2017-0-00195) and in part by the ITRC support program (IITP-2019-2017-0-01633) supervised by the IITP.

S. Han, S. Jang, H. Choi, H. Lee, and S. Pack are with the School of Electrical Engineering, Korea University, Seoul, Korea. (e-mail: {hs1087, imsoboy2, ghdfhrooo, ghcks1000, shpack}@korea.ac.kr, corresponding author: S. Pack).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/OJCOMS.2020.2990182, IEEE Open Journal of the Communications Society

process them. Each stage processes the extracted headers using match-action tables and sends them to the next stage. After all the processes at the pipeline stages are completed, the processed headers are sent to the deparser, which will combine the headers to reconstruct the packet.

B. P4

P4 is a high-level programming language that can define how programmable elements handle packets. In P4, the header information to be extracted and the extraction order need to be defined in the parser. To this end, the header format (e.g., header fields and their lengths) is declared in the P4 program. In addition, a parse graph, consisting of state nodes and state transition edges, is defined, and it specifies the order of extracting headers and the transition conditions. Specifically, the order and conditions for performing match-action tables (e.g., which conditions/fields are matched and what actions consisting of primitive actions are performed accordingly) are specified as a control flow, which is represented as a directed acyclic graph (DAG) called a control flow graph (CFG). Also, the deparser specifies how to shape the outgoing packet.

The P4 program written in the above manner is installed in P4-enabled switches through two-phase compilation with consideration of the P4-related constraints [14] such as target-independent and dependent constraints. In the first phase, the target-independent constraints (e.g., loop-free CFG derived from P4 language semantics) for a given P4 program are verified, and the program is converted into a high-level intermediate representation (HLIR). After that, in the second phase, HLIR is converted into a target-dependent program written in machine language with verifying the target-dependent constraints (e.g., memory size in the target machine). Once the P4 program is installed, the data plane can be managed via the P4 runtime application programming interface (API) that can populate the match-action table rules at runtime.

III. State of the Art on PDP Virtualization

In the literature, several schemes have been proposed for PDP virtualization using P4 and PISA, and they can be classified into 1) hypervisor-based approach [5], [6], [7] and 2) compiler-based approach [8], [9].

A. Hypervisor-based Approaches

Hypervisor-based approaches introduce a special purpose program that provides a platform on which multiple P4 programs can be installed. Depending on the language used for the program (i.e., P4 or hardware definition language (HDL)), the approaches can be further classified into 1) high-level hypervisor (denoted by P4-hypervisor) and 2) low-level hypervisor (denoted by HDL-hypervisor).

Figure 2 shows a reference architecture for P4-hypervisor, which includes a parser, match-action stages, and a deparser. Since P4-hypervisor is introduced to accommodate arbitrary P4 programs, its elements are defined to extract arbitrary types of headers and process arbitrary match-action tables instead of having a specific parser and match-action tables. Once

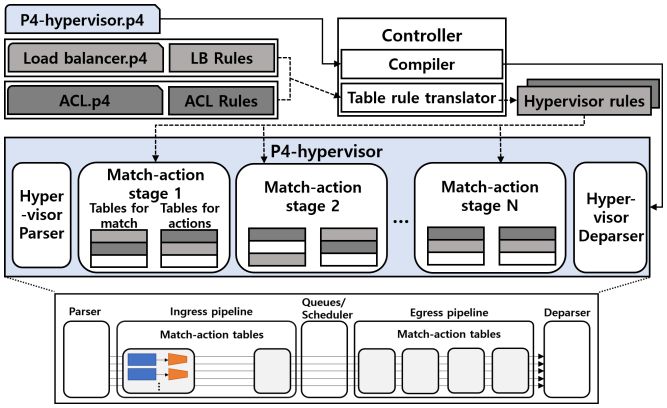

Fig. 2: P4-hypervisor architecture.

P4-hypervisor is installed, specific P4 programs (e.g., load balancer and firewall) can be loaded over P4-hypervisor. Since P4-hypervisor pre-configures generalized match-action tables for virtualization, the specific P4 program should be converted into table entries that can be populated on P4-hypervisor by means of a table rule translator. Hyper4 [5] and HyperVDP [6] are two representative hypervisor-based PDP virtualization schemes which use the high-level hypervisor. Both follow a similar architecture and operation process; however, different parsers and match-action stages are defined, which will be elaborated later.

Meanwhile, P4VBox [7] is a representative PDP virtualization scheme that uses HDL-hypervisor. This scheme supports virtualization by directly defining a parallel architecture for accommodating multiple programs and their behavior on a programmable device. A detailed description will be given later.

1) Hyper4: Hyper4 is the first study to virtualize PDP by means of P4-hypervisor. Its key idea is to extract a part of the header recursively until all the headers are extracted. In addition, a unique table to accommodate any type of matchings (e.g., exact and ternary) and primitive actions are introduced.

The workflow of Hyper4 is shown in Figure 3. First, the parser extracts the first N bytes from the header (steps 1-2). The extracted N bytes are then sent to the setup stage located in the ingress pipeline. In the setup stage, a program ID, which indicates the program to handle the incoming packet, is assigned based on the intrinsic metadata (e.g., ingress port) (step 3). After that, the setup stage checks whether the total header of the incoming packet is completely extracted. If the total header was not fully extracted, the parsing state table, which contains information about the header to be extracted, is evaluated to determine the value of N for the next extraction cycle (steps 4-5). After deciding the value of N, the packet is resubmitted from the end of the ingress pipeline to the parser and the above-mentioned procedures are repeated. Only when the total header is completely extracted can the packet be moved to the match-action stage. Also, in the setup stage, a match key is pre-specified in the metadata, which will be used to determine an appropriate match table (i.e., match table for header, intrinsic metadata, or user-defined metadata matching) (step 6). The determined match table returns an action ID that

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/OJCOMS.2020.2990182, IEEE Open Journal of the Communications Society

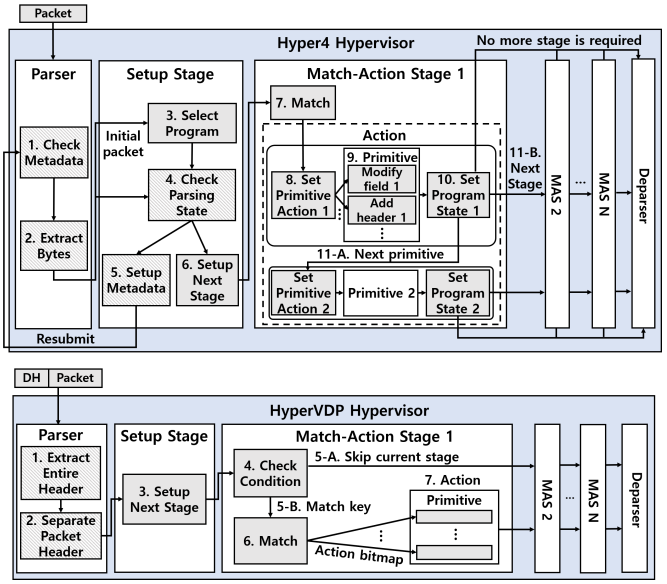

Fig. 3: Workflow of two hypervisor-based approaches.

indicates the compound action to be performed in the current match-action stage (step 7). Based on the action ID, the first primitive action is prepared (step 8), and it is executed using a corresponding table for executing the primitive action (step 9). After that, the parsing state check table inspects whether all the required primitive actions have been performed or not (step 10). If additional primitive actions need to be performed, the above-mentioned process is conducted with a new table for the next primitive action (step 11-A). Otherwise, the packet is transferred to the next match-action stage (step 11-B). Once all the match-action stages are finished, the processed headers are reassembled into the packet in the deparser and the packet exits P4-hypervisor.

2) HyperVDP: Compared with Hyper4, HyperVDP introduces a different parsing structure using a description header (DH), which includes additional information such as the program ID and the total header length. In addition, HyperVDP employs an improved match-action stage to reduce the number of steps in the action process.

As shown in Figure 3, the parser of HyperVDP extracts the entire header including the packet header and DH at once according to the total header length described in DH (step 1). After that, the packet header is separated from the entire header (step 2). This parsing procedure enables faster parsing because it extracts the packet header at once without any auxiliary processes to resubmit the packet as in Hyper4. After the parsing process and setup stage (step 3), the packet moves to the match-action stage. Unlike Hyper4, the match-action stage in HyperVDP can be skipped to avoid any unnecessary match-action processing. To this end, a condition check table, which examines the stage ID assigned during the setup stage and the program ID, is introduced (step 4). If no more match-action process is required in the current stage, the packet is directly sent to the next match-action stage (step 5-A); otherwise, the condition check table returns a match key (step 5-B) and an appropriate match table returns an action bitmap indicating a

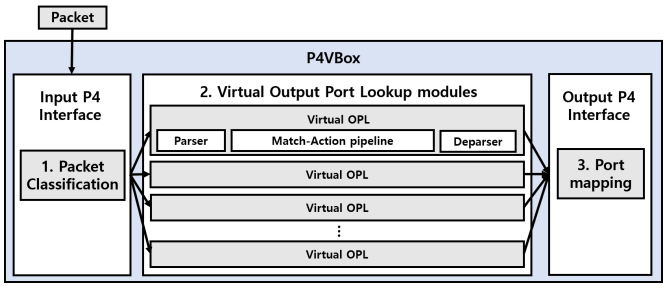

Fig. 4: P4VBox architecture.

set of primitive actions to be processed accordingly (step 6). Based on the action bitmap, the tables for executing primitive actions can be called directly and performed sequentially without preparing any primitive actions as in Hyper4 (step 7). The remaining procedures are similar to Hyper4, which are omitted for clarity.

3) P4VBox: Unlike Hyper4 and HyperVDP, P4VBox redesigns parallel lookup modules by means of HDL to support multiple programs on the data plane simultaneously. The architecture of the P4VBox is shown in Figure 4. Instead of using single output port lookup (OPL) in NetFPGA [15], P4VBox introduces 1) multiple virtual OPL modules, 2) input P4 interface (IPI) module, and 3) output P4 interface (OPI) module. When a packet arrives at P4VBox, the IPI module classifies the packet depending on its switch identier (Sid) and delivers the packet to the corresponding virtual OPL module (step 1). After that, each virtual OPL module processes the received packet according to the installed P4 program and the processed packet exits the virtual OPL module via its virtual output port (step 2). Note that a virtual OPL module acts as a programmable switch consisting of a parser, match-action pipelines, and a deparser. Finally, the OPI module lookups the port mapping table to connect the virtual port and the physical port, and forwards the processed packet to the next hop (step 3).

B. Compiler-based Approaches

Compiler-based PDP virtualization aims to provide a compiler-level virtualization layer to support the concurrent execution of multiple P4 programs on a single programmable switch. The main idea is to merge multiple P4 programs into a single P4 conguration file while preventing interference in terms of shared resource usage and functionality between P4 programs.

Figure 5 shows the architecture of the compiler-based approach that includes a P4 compiler, runtime agents, and a P4-enabled switch. The overall architecture is fairly similar to that of the conventional PDP as described in Section II; however, the compiler-based approach employs the P4 runtime agent with a merging context and a compiler with a composition function (i.e., function to merge multiple P4 programs) to enable PDP virtualization. The merging context represents the mapping information between the original program's and the merged program's components (i.e., programs, headers, tables identifiers), which is used for enforcing different control

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/OJCOMS.2020.2990182, IEEE Open Journal of the Communications Society

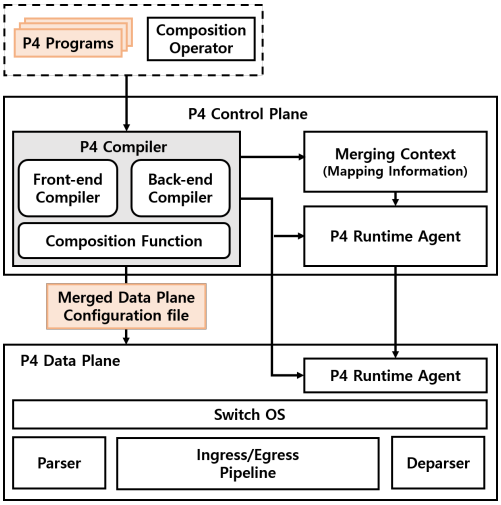

Fig. 5: Architecture of compiler-based PDP virtualization.

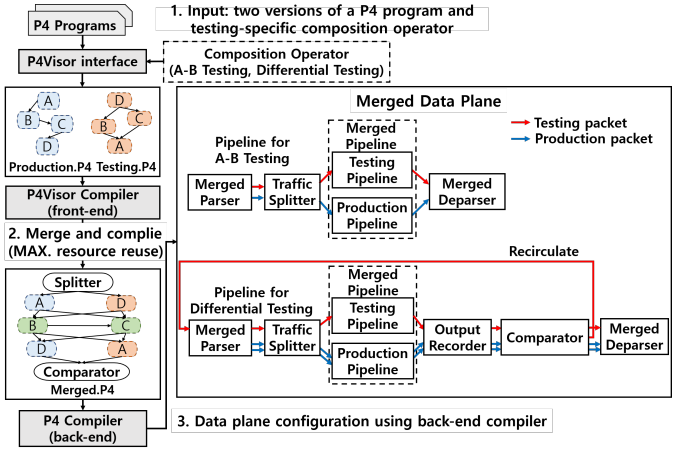

Fig. 6: Workflow of P4Visor.

policies over each data plane of the original P4 programs. Meanwhile, the roles of the composition function can be summarized as follows: 1) merging multiple parse graphs, 2) merging multiple CFGs, and 3) merging multiple deparse graphs. Since these graphs in a P4 program can be abstracted in the form of DAG, the merging operation is the same as solving a creation problem of the union of DAGs. Intuitively, the number of solutions for merging DAGs increases with the number of cycles in the union of the multiple graphs, and therefore the criteria such as constraints and processing policies should be clearly defined. This issue will be discussed for two representative compiler-based approaches (i.e., P4Visor [8] and P4Bricks [9]) in the following subsections.

1) P4Visor: P4Visor aims to support software testing (e.g., canary testing and data-differential testing) directly in the data plane, which is widely used in production networks for highly-available service deployments. The key function for the software testing is to run multiple versions of a program alongside each other, and therefore P4Visor provides compiler-level virtualization to allow their concurrent operations.

Figure 6 shows the workflow of P4Visor. First, P4Visor takes 1) two different versions of a P4 program, Production.P4 and Testing.P4, and 2) an A-B testing operator for canary testing or a differential testing operator for data-differential testing as the inputs (step 1). These composition operators give instructions to P4Visor compiler (PVC) for creating one merged P4 program with two logical pipelines in parallel while providing testing specific modules such as traffic splitter, output recorder, and comparator as shown in the merged pipeline. After receiving two versions of the P4 program and operators, PVC initiates a merging process (step 2). Specifically, PVC translates each P4 program into a form of DAG and identifies the components of each DAG. After that, PVC finds an identical state that can be potentially merged from each graph and merges two DAGs into one according to the composition operator and objectives. Since P4Visor focuses on the front-end compiler optimization to maximize the shared resources between different versions of P4 programs in the merging process, it provides a heuristic algorithm for efficiently solving the problem by taking target-independent constraints into consideration. For example, the generated Merged.P4 shows that PVC merges nodes B and C from two DAGs of Production.P4 and Testing.P4 to maximize the shared resources while maintaining target-independent constraints and applying packet processing policies (enforced by testing-specific operators). Finally, the output of P4Visor is consumed by the P4 back-end compiler to configure the data plane (step 3).

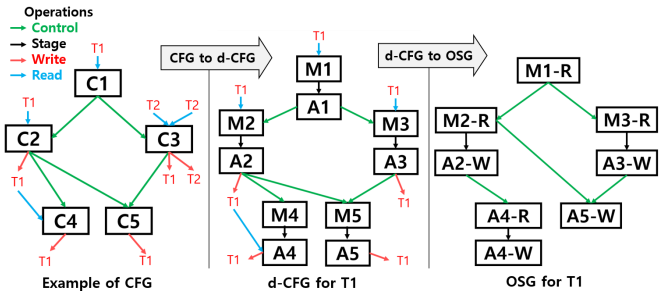

Fig. 7: Construction of OSG in P4Bricks.

2) P4Bricks: Unlike P4Visor, P4Bricks aims to create one data plane configuration file from multiple compiled data plane files with parallel and sequential composition operators. To this end, P4Bricks introduces a P4linker module for the composition function, which is similar to the linkage editor in the system programming field. While the merging of multiple P4 programs in P4Bricks is similar to that of P4Visor, the key difference is that P4Bricks additionally considers the operation scheduling while merging the CFGs.

Figure 7 illustrates the construction of an operation schedule graph (OSG) from CFG in P4Bricks. Recall that a P4 program can be defined to process multiple traffic types (e.g., headers, metadata); two traffic types, namely T1 and T2, are considered in Figure 7. P4linker extracts CFGs from the data plane configuration files and then decomposes each CFG into a decomposed-CFG (d-CFG) for each traffic type. Specifically, P4linker decomposes the control nodes of CFG into match (M) and action (A) nodes, and generates d-CFG for each traffic type. For example, a control node (e.g., C1 in Figure 7) is

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/OJCOMS.2020.2990182, IEEE Open Journal of the Communications Society

TABLE I: Comparison of PDP virtualization techniques.

	Hyper4	HyperVDP	P4VBox	P4Visor	P4Bricks
Traffic Isolation (Criteria)	Intrinsic metadata	ID field in DH	Sid tag	Tflag header	Intrinsic metadata
Live Reconfigurability (PDP program / Table rule)	Supported / Supported	Supported / Supported	Partially Supported / Supported	Not Supported / Supported	Not Supported / Supported
Resource Efficiency	Low	Low	Medium	High	High

decomposed into match and action nodes (e.g., M1 and A1 in Figure 7).

After that, P4linker generates an OSG from d-CFG for each traffic type (e.g., the OSG for the traffic type T1 in Figure 7). The OSG indicates the sequence of all operations (i.e., read (R) and write (W)) that should be applied to the traffic type. For each identical traffic type, P4linker applies composition operators on OSGs and creates a single OSG. These merged OSGs capture all possible sequences of operations performed on the traffic types under the given composition of P4 programs. After that, P4linker merges the d-CFGs of P4 programs according to the composition operator by adding the required dependencies across identical nodes of CFGs. The merged d-CFGs and OSGs provide the scheduling order constraints for the match/action CFG nodes and the operations performed by these nodes on the traffic types, respectively. By considering these constraints, P4Bricks devised a stage mapping algorithm that maps the match-action tables of merged d-CFGs to the physical pipeline stages in an orderly manner.

IV. Comparative Study

In this section, we analyze both hypervisor-based and compiler-based PDP virtualization approaches in terms of 1) traffic isolation, 2) live reconfigurability, and 3) resource efficiency. The key comparison results are summarized in Table I.

A. Traffic Isolation

Traffic isolation and processor isolation should be provided for fully isolated environments to run multiple P4 programs on one device without interference in terms of functionality and performance. Here, traffic isolation guarantees that the input traffic can access only the tables of its target P4 program while cannot access the tables of other P4 programs. Meanwhile, processor isolation means that processing units such as CPU and ALU are used so as not to affect the packet processing of other P4 programs. However, since the current PDP virtualization techniques only deal with traffic isolation, we analyze each approach regarding traffic isolation.

To ensure the traffic isolation, the traffic for a P4 program should not be able to access the resource that is exclusively dedicated to another P4 program, especially the match-action table. Such traffic isolation can be implemented by a packet tagging technique using controllable internal metadata. That is, each metadata carries information to identify the resource associated with each P4 program, and it can be used as a match key for each table to ensure traffic isolation. For example, Hyper4 and P4Bricks use intrinsic metadata such as ingress ports for identifying P4 programs. On the other hand, HyperVDP, P4VBox, and P4Visor use the ID field in the description header, the Sid tag, and the Tflag header, respectively, instead of intrinsic metadata. Therefore, HyperVDP, P4VBox, and P4Visor require increased bandwidth usage.

B. Live Reconfigurability

Live reconfigurability represents the ability to add/remove a P4 program and rules to/from PDP without affecting other P4 programs running on the same switch at runtime. This feature is useful for maintaining the states of each P4 program and providing flexible configuration.

To add or remove P4 programs at runtime, the P4-hypervisor-based approaches adopt the table rule translator. Since table entries created by the table rule translator can be added or removed without any additional compilations by means of the P4 runtime API, adding/removing P4 programs at runtime can be easily supported. On the contrary, the compiler-based approaches do not provide live reconfigurability. This is because the compiler-based approaches merge multiple P4 programs into one P4 program during compile-time and install the merged program on the target device. Thus, for adding or removing a P4 program, recompilation is indispensable, which may lead to loss of states in the data plane (e.g., loss of table entries, counter, meter, and register value). Meanwhile, P4VBox partially supports live reconfigurability. Since P4VBox has a fixed number of virtual OPLs, new P4 programs can be added up to the maximum number of virtual OPLs at runtime. However, if all virtual OPLs are being fully used, no more P4 programs can be added, i.e., live reconfigurability cannot be supported in this situation. To accommodate more P4 programs even under such situations, P4Box should be reinstalled to change the maximum number of virtual OPLs, which leads to the loss of data plane state as in the compiler-based approaches.

C. Resource Efficiency

Since the data plane has limited resources (i.e., memory and physical pipeline stages), it is necessary to use the limited resource efficiently in order to accommodate as many P4 programs as possible. Moreover, the resource usage for match-action tables affects the PDP performance [5]. To assess the resource efficiency, we analyze the numbers of tables 1) that should be declared for PDP virtualization and 2) that are used at runtime.

In Hyper4 and HyperVDP, the number of declared tables is determined by the number of stages and the supported primitive actions in P4-hypervisor. For a simple P4-hypervisor with only one stage, Hyper4 and HyperVDP declare a few hundreds and dozens of tables, respectively. This is because Hyper4 makes individual tables for all actions that are supported in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/OJCOMS.2020.2990182, IEEE Open Journal of the Communications Society

TABLE II: Number of tables used at runtime.

	P4 and P4VBox	Hyper4	HyperVDP	P4Visor
L2 switch	2	13	5	3
Firewall	3	22	8	4
L2 switch & Firewall	5	24	10	6

the given stage while HyperV dynamically loads actions with one action table as mentioned in Section III-A. Note that only a part of the declared tables is actually used in Hyper4 and HyperVDP, and hence their resource efficiency is not good.

Meanwhile, in P4VBox, a virtual OPL is assigned to a P4 program and the same number of tables as those of a conventional P4 program are deployed on a virtual OPL. Also, all the declared tables are used for packet processing at runtime. On the other hand, if a smaller number of P4 programs than the number of virtual OPLs are deployed, remaining virtual OPLs will not be used, which lead to degraded resource efficiency.

In the compiler-based approaches, the number of declared tables is determined by how many and what kinds of P4 programs are merged. In both P4Visor and P4Bricks, tables that can be shared between two P4 programs can be declared as one shared table. Therefore, fewer tables are required after the merging operations. Moreover, all the declared tables are used for packet processing at runtime and thus the compiler-based approaches provide high resource efficiency.

To evaluate the resource efficiency in a quantitative manner, we implemented Hyper4, HyperVDP, and P4Visor on behavioral model version 2 (BMv2) [16] software switch and installed two P4 programs (i.e., L2 switch and firewall). [1] As shown in Table II, it can be found that Hyper4 and HyperVDP require increased numbers of tables of 24 and 10, respectively. Meanwhile, P4VBox and P4Visor need only 5 and 6 tables, respectively, for running two P4 programs simultaneously. Note that P4VBox does not need any additional tables compared to the original P4 whereas P4Visor uses one more table than the original P4 owing to the use of the table splitter. To summarize, hypervisor-based approaches using the P4-hypervisor can lead to poor resource efficiency, which should be addressed by further study.

V. Open Challenges for PDP Virtualization

In this section, we discuss the open challenges and opportunities to provide complete PDP virtualization and improve the performance and flexibility in the existing approaches.

A. Performance Isolation

Performance isolation means that P4 programs that share the resources of a physical device (i.e., memory and processor) operate in an isolated manners, and thus the performance of a P4 program is not affected by another P4 program. However, the current PDP virtualization schemes only provide memory isolation for match tables (i.e., traffic isolation) as described in Section IV-A; they do not support any processor isolation.

[1]Since P4Bricks is still an on-going work and there is no published implementation, we do not include it in our study.

TABLE III: Packet processing latency (ns).

	P4	HyperVDP	P4Visor
L2 switch	289	354	331
Firewall	301	354	331

To investigate processor isolation issues in more detail, we have measured the packet processing latency from the ingress pipeline to the egress pipeline for L2 switch and firewall P4 programs. For accurate measurements, we used a hardware switch using Tofino [18] (i.e., Wedge100BF-32X) and modified HyperVDP and P4Visor to be run on the hardware switch.

When either L2 switch or firewall is loaded in the conventional P4, firewall shows longer latency than L2 switch since it requires more physical stages, as shown in Table III. On the other hand, it can be found that L2 switch and firewall have the same latency when they are simultaneously loaded in HyperVDP and P4Visor. This is because the processing resources are shared by two P4 programs in HyperVDP and P4Visor (i.e., processor isolation is not supported) and the overall latency is determined by the P4 program with the longest processing latency.

To address this issue, a novel and programmable switch architecture needs to be devised. Chole *et al.* [3] proposed dRMT, which disaggregates the memory and the computing resource in the switch architecture. By means of such disaggregation, a packet can be processed entirely by one processing unit and the other packets can be processed with different processing units. In addition, dRMT resolves its architectural conflict problems such as simultaneous allocation of match operations beyond the capacity of the target hardware by means of efficient scheduling. However, since this scheduling affects the processing delay between processors, further study is required for achieving full isolation among processing units.

B. High Performance and Flexibility

Current PDP virtualization schemes have limitations in terms of performance and flexibility. First of all, the hypervisor-based approaches using the P4-hypervisor experience performance degradation due to several reasons. As discussed in Section IV, they require excessive table resources to handle arbitrary types of packets and actions. Specifically, they are designed to perform a single match-action with several steps as mentioned in Section III-A, and thus they use more physical stages. As a result, some packets can be recirculated to the ingress at the end of the pipeline to complete its tasks under given physical resources. This recirculation degrades overall performance significantly because packets pass through the pipeline once more. Consequently, to save the table resources and prevent frequent recirculations, the architecture of P4-hypervisor needs to be improved. Also, the hypervisor-based approaches use an entire packet header as a match key due to their unique parsing structures. Such a large match key causes increased memory access time and adversely affects the overall performance. Therefore, a new parsing structure to extract arbitrary headers while reducing the match key size needs to be defined.

On the other hand, the hypervisor-based approach using the HDL-hypervisor and compiler-based approaches can provide improved data plane resource usages, but they lack flexibility for live reconfiguration. Specifically, P4VBox requires full reconfiguration when more virtual OPLs are required to support an increased number of P4 programs and P4Visor and P4Bricks require frequent recompilations to add or remove a new program, which causes a loss of states for PDP. To address this issue, an appropriate state migration scheme is needed. Luo *et al.* [10] proposed a state management framework that allows the data plane states to be preserved during reconfiguration. Although the state management framework cannot support live reconfiguration, it can improve the flexibility by mitigating the loss of states during data plane reconfiguration.

VI. Conclusion

In this article, we provided an overview of P4 and PISA, which are representative of the language and architecture for PDP, respectively. After that, we categorized the PDP virtualization schemes into hypervisor- and compiler-based approaches, and discussed ways to achieve PDP virtualization along with their structural differences. In addition, their salient features were analyzed quantitatively and qualitatively in terms of traffic isolation, live reconfigurability, and resource efficiency. Also, we discussed the open challenges for PDP virtualization and their possible solutions. Since PDP virtualization is in its initial stage, we hope that this article provides a good starting point to design better PDP virtualization frameworks.

References

[1] B. Chung, C. Tseng, J. Chen, and J. Mambretti, "P4MT: Multi-Tenant Support Prototype for International P4 Testbed," in *Proc. ACM ANCS 2019*, Sep. 2019.

[2] Fabric-testbed. [Online]. Available: https://fabric-testbed.net/, accessed on Mar. 30, 2020.

[3] S. Chole, A. Fingerhut, S. Ma, A. Sivaraman, S. Vargaftik, A. Berger, G. Mendelson, M. Alizadeh, S. Chuang, I. Keslassy, A. Orda, and T. Edsall, "dRMT: Disaggregated Programmable Switching," in *Proc. ACM SIGCOMM 2017*, Aug. 2017.

[4] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeow, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, "P4: Programming Protocol-Independent Packet Processors," *ACM SIGCOMM Computer Communication Review*, Vol. 44, No. 3, pp. 8795, Jul. 2014.

[5] D. Hancock and J. Merwe, "Hyper4: Using p4 to Virtualize The Programmable Data Plane," in *Proc. ACM CoNEXT 2016*, Dec. 2016.

[6] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "HyperVDP: High-Performance Virtualization of the Programmable Data Plane," *IEEE Journal on Selected Areas in Communications*, Vol. 37, No. 3, pp. 556-569, Mar. 2019.

[7] M. Saquetti, G. Bueno, W. Cordeiro, and J. Azambuja, "P4VBox: Enabling P4-Based Switch Virtualization," *IEEE Communication Letters*, Vol. 24, No. 1, pp. 146-149, Jan. 2020.

[8] P. Zheng, T. Benson, and C. Hu, "P4Visor: Lightweight Virtualization and Composition Primitives for Building and Testing Modular Programs," in *Proc. ACM CoNEXT 2018*, Dec. 2018.

[9] H. Soni, T. Turletti, and W. Dabbous,, "P4Bricks: Enabling Multiprocessing Using Linker-based Network Data Plane Architecture," HAL arXiv, https://hal.inria.fr/hal-01632431, Feb. 2018.

[10] S. Luo, H. Yu, and L. Vanbever, "Swing State: Consistent Updates for Stateful and Programmable Data Planes," in *Proc. ACM SOSR 2017*, Apr. 2017.

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

[12] 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 *Proc. ACM SIGCOMM 2017*, Aug. 2017.

[13] X. Jin, H. Zhang, R. Soul, J. Lee, N. Foster, C. Kim, and I. Stoica, "NetCache: Balancing Key-Value Stores with Fast In-Network Caching," in *Proc. ACM SOSP 2017*, Oct. 2017.

[14] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling Packet Programs to Reconfigurable Switches," in *Proc. USENIX NSDI 2015*, May 2015.

[15] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4→NetFPGA Workflow for Line-Rate Packet Processing," in *Proc. FPGA 2019*, Feb. 2019.

[16] Behavioral model version 2 (BMv2). [Online]. Available: https://github.com/p4lang/behavioral-model, accessed on Aug. 22, 2019.

[17] Protocol-Independent Switch Architecture (PISA). [Online]. Available: https://barefootnetworks.com/white-paper/the-worlds-fastest-most-programmable-networks/, accessed on Mar. 30, 2020.

[18] Barefoot's Tofino, [Online]. Available: https://barefootnetworks.com/products/brief-tofino/, accessed on Mar. 30, 2020.

Sol Han (hs1087@korea.ac.kr) received the B.S. degree from Korea University, Seoul, Korea, in 2015. He is currently an M.S. and Ph.D. integrated course student in School of Electrical Engineering, Korea University, Seoul, Korea. His research interests include SDN/NFV and P4 programmable networking.

Seokwon Jang (imsoboy2@korea.ac.kr) received the B.S. degree from Korea University, Seoul, Korea, in 2015. He is currently an M.S. and Ph.D. integrated course student in School of Electrical Engineering, Korea University, Seoul, Korea. His research interests include SDN/NFV and P4 programmable networking.

Hongrok Choi (ghdfhrooo@korea.ac.kr) received the B.S. degree from Korea University, Seoul, Korea, in 2018. He is currently an M.S. and Ph.D. integrated course student in School of Electrical Engineering, Korea University, Seoul, Korea. His research interests include network traffic analysis and P4 programmable networking.

Hochan Lee (ghcks1000@korea.ac.kr) received the B.S. degree from Korea University, Seoul, Korea, in 2018. He is currently an M.S. and Ph.D. integrated course student in School of Electrical Engineering, Korea University, Seoul, Korea. His research interests include intelligent network management and P4 programmable networking.

Sangheon Pack [SM] (shpack@korea.ac.kr) received the B.S. and Ph.D. degrees from Seoul National University, Seoul, Korea, in 2000 and 2005, respectively, both in computer engineering. In 2007, he joined the faculty of Korea University, Seoul, Korea, where he is currently a professor in the School of Electrical Engineering. His research interests include future softwarized networking (SDN/NFV) and mobile cloud networking/edge computing.