P4DB: On-the-fly Debugging for Programmable Data Planes

Yu Zhou, Jun Bi, *Senior Member, IEEE, ACM,* Cheng Zhang, Bingyang Liu,
Zhaogeng Li, Yangyang Wang, Mingli Yu

***Abstract*—While extending network programmability to a more considerable extent, P4 raises the difficulty of detecting and locating bugs, *e.g.,* P4 program bugs and missed table rules, in runtime. These runtime bugs, without prompt disposal, can ruin the functionality and performance of networks. Unfortunately, the absence of efficient debugging tools makes runtime bug troubleshooting intricate for operators. This paper is devoted to on-the-fly debugging of runtime bugs for programmable data planes. We propose P4DB, a general debugging platform that empowers operators to debug P4 programs in three levels of visibility with rich primitives. By P4DB, operators can use the `watch` primitive to quickly narrow the debugging scope from the network level or the device level to the table level, then use the `break` and `next` primitives to decompose match-action tables and finely locate bugs. We implement a prototype of P4DB and evaluate the prototype on two widely-used P4 targets. On the software target, P4DB merely introduces a small throughput penalty (1.3% to 13.8%) and a little delay increase (0.6% to 11.9%). Notably, P4DB almost introduces no performance overhead on Tofino, the hardware P4 target.**

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

I. Introduction

PROGRAMMABLE data planes (PDP) together with domain-specific languages, *e.g.,* P4 [2] and POF [3], unleash a new wave of programming networks. Recent researches [4]–[9] present a growing and promising trend that more and more sophisticated network functions can be implemented as P4 programs to run on programmable switches, which significantly improves performance and flexibility of networks.

P4 provides various programmable elements for operators to *flexibly* define packet-processing behaviors (*i.e.,* forwarding ports and header modification) on PDPs. Firstly, operators can customize parsers to extract header fields with self-defined protocol formats. Secondly, for match-action tables (MAT), operators can define various match fields and construct compound actions with primitive actions. Thirdly, operators can organize multiple MATs as a consolidated Directed Acyclic Graph (DAG) in the control flow. Lastly, operators can declare data plane variables, *e.g.*, metadata, registers, to store flow states and conduct stateful operations. At runtime, the centralized controller, such as P4Runtime [10] and P4-ONOS [11], can control a P4 program through populating MAT entries. By provisioning an intuitive programming abstraction based on above programmable elements, P4 empowers operators to conduct in-depth customization of their networks, which enables fast innovations on network infrastructures and protocols.

With PDP and P4 growing in importance and maturity, *runtime bugs* in PDPs are becoming critical concerns. In particular, PDP runtime bugs include P4 program bugs, misconfigured MAT entries, inconsistency between P4 targets and P4 specifications, implementation-specific bugs in P4 targets, and so on. Runtime bugs could cause abnormalities (*e.g.,* silent packet drops) in running P4 targets and might be only triggered by specific flows under particular conditions. Although troubleshooting runtime bugs is of great importance to guarantee the correctness of PDPs, no debugging tool for real P4 devices is available. Thus operators have to debug them manually, which is cumbersome and time-consuming.

Furthermore, the following challenges make detecting and locating runtime bugs for PDPs a difficult task for operators.

C_1: Diversity of runtime bugs. Runtime bugs may happen to any programmable element in diverse styles. For example, operators may mistake a wrong field for the right one when defining the if-else expressions, which makes packets traverse undesired MATs. Moreover, a P4 target may not correctly implement the primitive actions (as is shown in §IV-B), and those primitive actions do not produce the behaviors defined by the P4 specification. Accordingly, these bugs can cause diverse kinds of abnormalities, *e.g.*, malformed packets, black holes, and permanent loops, which further hinder operators from finding out causes of the runtime bugs.

C_2: Invisibility of intrinsic states. Once a P4 program is deployed onto a P4 target, operators cannot watch intrinsic states that record processing behaviors of each packet in PDPs. For P4, intrinsic states include intermediate metadata, traversed MATs, primitive action parameters, and so on. These intrinsic states are vital for detecting and locating runtime bugs. For example, if operators cannot directly observe the traversed MAT path (*i.e.,* an ordered list of MATs that are traversed by packets) of packets, operators cannot find which

This work is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (No.61472213). Yangyang Wang is the corresponding author.

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

Yu Zhou, Cheng Zhang and Zhaogeng Li, Mingli Yu are with the Institute for Network Sciences and Cyberspace, Tsinghua University, the Department of Computer Science, Tsinghua University, and the Beijing National Research Center for Information Science and Technology (BNRist) (e-mail: {y-zhou16, cheng-zhang13, li-zg07, yml14}@mails.tsinghua.edu.cn).

Bingyang Liu is with Huawei Technologies Co., Ltd. (liubingyang@huawei.com).

A previous version of this paper has been published at ICNP'17 [1].

MAT causes packet dropping (*e.g.,* due to access controlling or incorrect forwarding rules). However, P4 targets do not provide sufficient access to the intrinsic states, which inevitably makes debugging PDPs at runtime a challenging task.

C_3: Complexity of locating bugs. As P4 programs grow in size, operators can be overwhelmed by the complexity of troubleshooting runtime bugs. For example, *Switch.P4* [12] contains 129 MATs, 76 *if-else* expressions, and 340 compound actions. Thus, when an operator wants to find out why *Switch.p4* does not forward a packet as expected, he can do nothing but dumps all table entries and reasons the bugs MAT by MAT. Moreover, target-specific bugs further add up the debugging complexity. For example, if a P4 target silently drops packets when a MAT has an oversized match vector (as is shown in §IV-B), dumping table entries and reasoning P4 programs cannot help locate the bug.

Unfortunately, there remains a gap between existing debugging tools and satisfactory solutions to the above challenges. On the one hand, existing tools are mainly designed for well-studied traditional networks and can hardly handle runtime bugs on PDPs. These tools can be categorized into the following two types. Firstly, debugging tools [13]–[20] are based on monitoring techniques. They focus on modelling MATs without consideration of other programmable elements. Besides, they suffer from a large performance overhead of tracing real-time network traffic. Secondly, other debugging tools [21]–[24] are based on static verification techniques. They cannot automatically model the changes to forwarding behaviors without reprogramming the model internals [25]. Moreover, they usually hypothesize that the data plane model is correct and verify network properties (*e.g.,* loop and reachability) based on a snapshot of network states (*i.e.,* MAT entries). They cannot troubleshoot runtime bugs on PDPs.

On the other hand, *verifying* P4 programs raises new researching interests and comes with a line of new works [25]–[29]. These works can debug P4 programs and network state snapshots through formal methods, *e.g.,* Symbolic Execution [30]. With these tools, operators can detect and locate a subset of runtime bugs such as bugs in P4 programs and misconfigured MAT entries, but these P4 verification works can hardly support real-time troubleshooting due to *long execution time*. As these works cannot provide visibility into running P4 targets, they *fall short of disposing of some runtime bugs*, *e.g.,* inconsistency between P4 targets and the P4 specification and wrong primitive action implementation in P4 targets.

To bridge the gap and efficiently debug PDPs in runtime, this paper introduces P4DB, a general debugging platform for PDPs. By P4DB, operators can flexibly scrutinize PDPs and P4 programs in real time with *three levels of visibility*.

- *Network-level visibility* provides operators with the network path of the specified flow.
- *Device-level visibility* provides operators with the MAT path of the specified flow in a running P4 program.
- *Table-level visibility* enables operators to inspect the packet processing procedure step by step inside the MAT.

To support the visibility and simplify the debugging workflow, P4DB provides three novel designs.

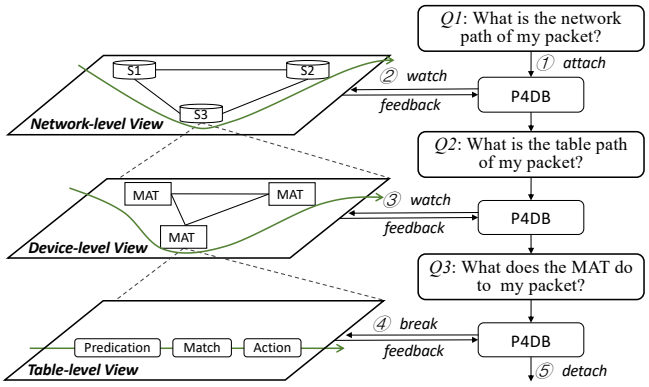


Fig. 1. Debugging the P4-enabled network with P4DB.

$\mathcal{D}_1$: Debugging snippets are designed to reveal real-time states of programmable elements in PDP (C_1 & C_2). A debugging snippet, in essence, is a data-plane code piece that can be flexibly inserted into specified positions in a P4 program and report desired intrinsic states at runtime.

$\mathcal{D}_2$: Three-step MAT decomposition is devised to decompose the MAT along with its predication expression equivalently into three steps including predication step, match step and action step (C_1 & C_2). Therefore, operators can embed debugging snippets between these steps to finely inspect the execution of the MAT in a single-step way.

$\mathcal{D}_3$: Debugging primitives are implemented by different compositions of debugging snippets in PDP and provide intuitive and usable interfaces for operators to facilitate debugging workflow (C_3). With the help of the debugging primitives including watch, break, next, *etc.*, operators can avoid dumping table entries and reasoning the bugs out table by table. Based on these primitives, operators can develop debugging tools that automatically troubleshoot the runtime bugs.

As is shown in Fig. 1, when debugging P4 programs, operators can use the watch primitive to narrow the debugging scope quickly from the network level to the table level. Then, operators can use the break and next primitives to decompose the MAT and inspect what happens to the packet at runtime. We employ the debugging primitives to implement four debugging tools including the *network debugging CLI*, the *flow path monitor*, and so on, which are elaborated in §II-C.

In this paper, our contributions are as follows:

- To the best of our knowledge, P4DB is the first design of an on-the-fly debugging platform to troubleshoot runtime bugs in PDP. We demonstrate the *feasibility* and *versatility* of P4DB through real-world examples.
- For P4DB, we propose three novel designs to facilitate troubleshooting various runtime bugs, simplify the debugging workflow, and provision different levels of visibility.
- We implement the prototype of P4DB based on two recently-proposed PDP models: the P4-specific PDP model and the hypervisor-specific PDP model. Accordingly, we evaluate P4DB on various P4 targets. Results indicate that P4DB merely introduces a small performance overhead on the software target and almost introduces no performance overhead on the hardware target.

The remainder of this paper is organized as follows. We firstly describe the philosophy and functionality of P4DB in §II. After that, the key designs are shown in §III. In §IV, the debugging workflow of P4DB is demonstrated through a real-world example, and we will introduce our experiences of employing P4DB to troubleshoot runtime bugs on two widely-used P4 targets. In §V, we evaluate P4DB. Then, we discuss related work in §VI. Finally, we discuss feasibility of P4DB in §VII and make a conclusion in §VIII.

II. Overview of P4DB

A. The Philosophy of P4DB

Most bugs arise from the mistakes and errors made in either a program's source code or its design. Therefore, an efficient way to debug is to allow programmers to inspect what is happening inside the program. For operating systems, programmers can load the application's source code into the debugger such as GDB [31], then insert a breakpoint into the source code and check what is happening at runtime. When a developer sets a breakpoint for a program, a small piece of trap code, which switches the execution subject from the program to the debugger, is inserted into the program. However, in legacy networks, network devices are black boxes, and operators have no access to the source code, which makes the data plane hard to be debugged like software. Compared with the legacy networks, PDP enables operators to flexibly program device behaviors, which makes it possible to trace real-time data plane states and debug data planes with an approach similar to GDB.

Since no trap instruction is available for operators to stop the execution of P4 programs, P4DB introduces pieces of "report code", named *debugging snippets*, to report the states of programmable elements. A debugging snippet is a small control flow composed of several MATs and can report on-data-plane states of user-interested flows with a minor performance overhead. Besides, different combinations of debugging snippets can be used to implement various user-friendly debugging primitives for operators. So operators can use debugging primitives to dynamically control the debugging snippets in the program and debug the program at runtime. This way of using debugging snippets to implement debugging primitives makes P4DB a general debugging platform which is not constrained by the underlying targets and can be applied to various MAT-based architectures including RMT [32] and dRMT [33]. Furthermore, our design does not require any modification of P4 targets, which implies that P4DB can work smoothly with existing P4 targets.

B. PDP Models Supported by P4DB

P4DB is a general debugging platform to work with various PDP models. In this section, we briefly introduce two available PDP models and their characteristics. Next, we describe some model-specific features utilized by P4DB.

P4-specific PDP model. The P4-specific PDP model is a widely adopted model. In this PDP model, the programming abstraction used in the P4 language is tightly coupled with the P4 target implementation. Therefore, P4 programs can fully benefit from the high performance of hardware devices [34]. However, this tight coupling between the high-level program and the low-level hardware device can lead to a result that every time operators need to modify the P4 program, the interruption and reconfiguration of the P4 target are unavoidable.

Hypervisor-specific PDP model. Novel PDP models such as Hyper4 [35], MPVisor [36], and HyperV [37], are proposed to decouple high-level P4 programs with low-level P4 targets by adding a light-weight hypervisor layer. Although there is a performance overhead due to the additional hypervisor, operators can dynamically reconfigure P4 programs without suspending the P4 target. The hypervisor-specific PDP model tries to provide the same programming abstraction for operators with the P4-specific PDP model, *i.e.,* supporting direct execution of native P4 programs. The key difference between the hypervisor-specific PDP model and the P4-specific one is that the former one supports dynamically loading P4 programs.

Features on various PDP models. Both the P4-specific PDP model and the hypervisor-specific PDP model have their pros and cons. P4DB is designed as a debugging platform that presents debugging features by utilizing the programmability provided by various PDP models. Meanwhile, P4DB also offers the unique features of multiple PDP models as special debugging features when being applied to debug the specific PDP model. For instance, based on the hypervisor-specific PDP model, P4DB can benefit from the uninterpreted feature, i.e., P4DB does not need to redeploy the model to install new debugging snippets, while it suffers from a performance overhead of model translation. Oppositely, for the P4-specific PDP model, P4DB provides high performance and language compatibility but needs to recompile and redeploy the debugged P4 program when installing new debugging snippets. The detailed implementation of P4DB upon various PDP models is further described in §III-H.

C. Debugging Tools Built upon P4DB

To demonstrate what debugging abilities P4DB provides and how P4DB simplifies the workflow of debugging runtime bugs, we introduce four useful debugging tools implemented by the debugging primitives of P4DB.

Network debugging CLI. This tool shares a similar functionality with ndb [13] but provides more feature-rich debugging commands than ndb. Specifically, *network debugging CLI* provides a native implementation for the high-level primitives of P4DB. With the CLI, operators can use the useful commands to debug P4 programs interactively.

Flow path monitor. Based on the watch primitive, the flow path monitor provides the real-time information about which path the packets of a designated flow traverse. Operators can attain one sequenced port list which comprises the ingress port and egress port of every traversed switch. Notably, based on the network-level visibility of P4DB, the *flow path monitor* can easily handle the complicated condition where the flow has multiple paths.

Network-wide invariant verifier. This tool is based on the *flow path monitor* to verify two network-wide invariants including *reachability* and *loop-free*. Based on the path information collected by the monitor, the *network-wide invariant*

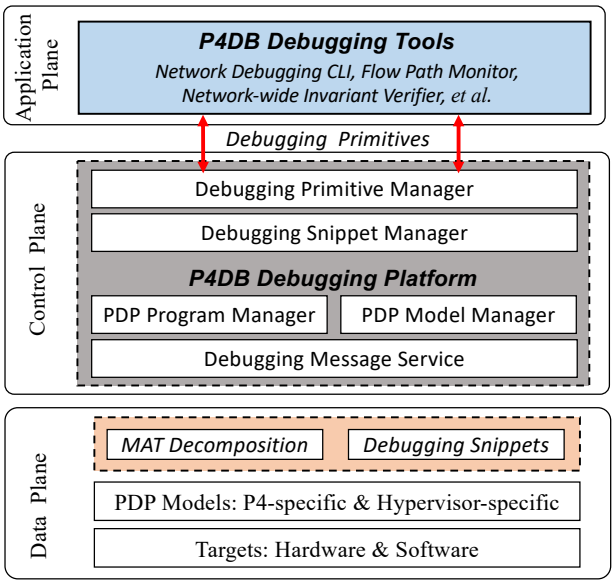


Fig. 2. Architecture of P4DB.

verifier can easily check whether the flow is correctly forwarded. As the verifier can get the ingress ports and egress ports of the packets from the monitor, it can verify reachability and loops for every individual path of the flow without any confusion. Besides, the verifier can be configured to emit invariant violation alerts automatically or to trigger the failover operations such as recalculating forwarding paths.

Security policy checker. The objective of the *security policy checker* is to check whether security-related functions (e.g., firewall) correctly process packets according to security policies. Unlike the *flow path monitor* which only requires the watch primitive, the *security policy checker* needs to cooperate with the break and next primitives to check every MAT of security-related functions. The checker can not only report the security policy violation events but also help locate the misconfigured MATs, which enables operators to resolve the security policy violation events timely.

III. Design of P4DB

A. System Architecture

Fig. 2 shows the architecture of P4DB. Operators can develop versatile debugging tools with the debugging primitives. The debugging primitives are implemented by different compositions of debugging snippets. The PDP program manager maintains the status of all P4 programs and their running instances distributed in networks. The PDP model manager maintains the mapping of the PDP model and the hardware devices. The debugging message service maintains the control channel between debugging snippets and the debugging platform. Next, we will respectively elaborate the three primary techniques including the *three-step MAT decomposition*, the *debugging primitives*, and the *debugging snippets*.

B. Three-step MAT Decomposition

To provide fine-grained visibility, we decompose one MAT into three sequential steps, *i.e.,* the predication step, the match step, and the action step. The predication step is functionally equivalent to *if-else* expressions bounded with the MAT in the control flow. Similarly, the match step executes the match logic of the original MAT and outputs the table entry index. The action step executes the action logic. Based on the MAT decomposition, P4DB inserts the snippets after each corresponding step to collect the data plane states and to analyze data plane states for each step respectively. Then, operators can use next to verify the correctness of each step.

Reasons for decomposing MATs. In P4 programs, the results of the if-else expression, compiled like "hard-coded logic" between MATs, are invisible to operators at runtime. Thus, if the expression is wrongly programmed, there is no efficient way to identify the bug except for manually reasoning packet behaviors, which is nearly infeasible for real networks. As for the match step and the action step, they are tightly coupled with each other and concealed in one MAT. Consequently, operators can only review the results for the MAT rather than intrinsic states of the MAT. In the real-world debugging case, knowing (*i*) which fields are matched and (*ii*) which actions are executed for this packet are essential for operators to debug the MAT. Thus, P4DB decomposes the MAT into three steps and enables operators to finely oversee which actions are applied to the exact packet in the MAT.

Simulation of Single-step Debugging. Although one MAT can be decomposed into three steps, P4DB cannot suspend the processing of a packet and conduct the decomposed steps one by one for the packet. Thus in P4DB, we design a simulative way with an assumption that packets in the network can trigger the bug repetitively. In operating systems, the programmer can merely observe the occurrence of the bug only after the bug is triggered. However, the runtime bug in a P4 program can be triggered by millions of packets belonging to the same flow. In other words, there is no need for operators to follow the procedure of processing one packet to find out what is happening. Instead, P4DB focuses on the P4 program itself and views packets as bug triggers. As long as the bug can be persistently triggered by the packets in the same flow, the single-step debugging approach can work well and enable operators to inspect what happens to packets step by step. Furthermore, we discuss bug recurrence in §VII.

C. Debugging Primitives

As shown in Fig. 3, we design a suite of high-level debugging primitives. Internally, debugging primitives are implemented by various compositions of debugging snippets. Until now, there are seven fundamental primitives. With the flourishing of debugging snippets, more debugging primitives will be developed to facilitate the debugging workflow.

The attach and detach primitives can dynamically attach/detach the debugger to a P4 program running on a designated device. When operators issue the attach primitive, P4DB loads the P4 program, analyzes the source code, checks the specific PDP model on the hardware device, and prepares the debugging environment.

The watch primitive provides operators with two levels of visibility. One is network-level visibility, which shows the

Primitives Synopsis

1) **attach** *program switch*
This primitive attaches the debugger to the P4 program on user specified switch.
program :
The name of the P4 program.
switch :
The identifier of the switch, usually the data path id.
use case:
\> **attach** myfirewall 100001

2) **detach** *program*
This primitive detaches the debugger from the attached program.
program :
The name of the P4 program.

3) **break** [-n *name*] –t *tablename* –f {[*field* : *value*], … }
This primitive sets a breakpoint to the match-action table specified by *tablename*, and chooses the flow specified by the set of *field* and *value* pairs to be debugged. The next primitive always follows the break primitive.
tablename :
The name of the table that is being debugged.
name :
The optional name of the breakpoint defined by user.
field :
The name of the field. A field can be any field declared in the P4 program, e.g. a header field or a metadata.
value :
The specific value of the field.
use case:
\> **breakpoint** –n mybp –t table_ipv4 –f {eth_hdr:0x1f 2f3f4f, ip_src:192.0.0.1, ip_dst:192.0.0.2}

4) **next**
This primitive will go through one step when the break primitive is triggered. Each next command will illustrate the information of predication, match and action step respectively.

5) **watch** [-s *switch*] –f {[*field* : *value*], … }
This primitive will illustrate the path information of a flow specified by the condition set of field and value.
switch :
If the switch is specified, watch primitive will show the table path traversed by the specific flow, otherwise, it will show the network path traversed by the specific flow.
field :
The name of the field. A field can be any field declared in the P4 program e.g. a header field or a metadata.
value :
The specific value of the field.
use case:
\> **watch** –s 100001 –f {ip_src:192.168.0.0.1, user_met a_var:2}

6) **show** [-b] [-p] [-a] [-t] [-h]
This primitive shows the information of corresponding object.
-b:
This option shows the information of all breakpoints.
-p:
This option shows the information of all programs.
-a:
This option shows the information of all compound actions.
-t:
This option shows the information of all tables.
-h:
This option shows the information of all header fields declared in the P4 program.

7) **rmbp** *name*
This primitive removes the breakpoint specified by the *name.*
use case:
\> **rmbp** mybp

Fig. 3. High-level debugging primitives provided by P4DB.

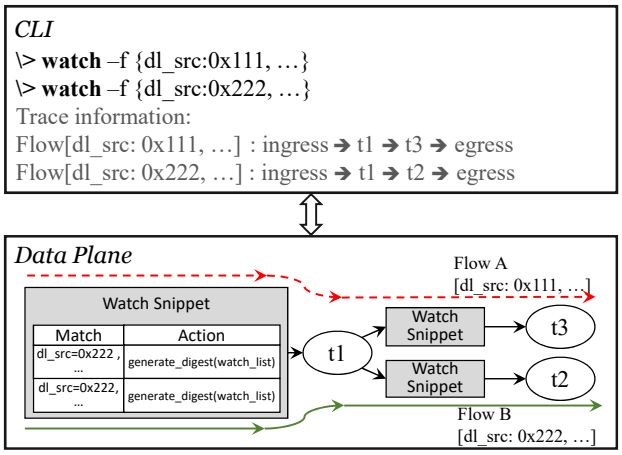


Fig. 4. Use the CLI to manipulate watch snippets.

network path of the specified flow. The other is the device-level visibility, which provides the MAT path of the specified flow in a P4 target. Internally, P4DB inserts some watch snippets into the switch or all switches, collects the reports, and shows the trace of the flow.

The break and next primitives together enable operators to debug one MAT with the table-level visibility. When operators issue the break primitive, P4DB decomposes the MAT and inserts the break snippet. Once the break snippet is triggered by the specified flow, operators can issue the next primitive to let P4DB dynamically install the predication snippet (or activated the preset snippet) into the decomposed MAT. Then operators can observe the states of the if-else expressions. Afterwards, the following two next primitives will respectively install the match snippet and action snippet, and present states of the match step and action step.

We omit other primitives (e.g., rmbp and show in Fig. 3) which are too trivial to be described in the paper. By utilizing these primitives, debugging PDP becomes much simpler. Operators can select a flow, get the network-wide trace of the flow, locate the possible device with bugs, check the device-wide trace of the flow, find the possible MAT with bugs, inspect the execution of the MAT step by step, and finally find the cause of the bugs. Through these debugging primitives, operators can conveniently debug the wrong implementations of various programmable elements, such as the *if-else* expressions, match operations, and compound actions. However, debugging the other programmable elements, such as the parser and deparser, exceeds the power of the debugging primitives.

D. Debugging Snippets

The debugging snippets match the flow specified by operators and report real-time states of programmable elements to the P4DB platform. The match rules in debugging snippets are instantiated by P4DB based on the parameters carried in debugging primitives. The actions in debugging snippets are set to report different programmable elements according to the types of debugging snippets. When operators issue a debugging primitive, P4DB will install the corresponding debugging snippet into the PDP program and populate the MATs in the debugging snippet. Different debugging snippets

report different digests. Once a debugging snippet is deployed in PDP, it can match the flow and use the *generate_digest* action to send reporting messages to the P4DB platform. There are five types of debugging snippets as follows.

Watch snippet. The watch snippet is implemented by one MAT. The match rules in the watch snippet are configured according to the parameters in the watch primitive. The actions in the watch snippet are set to report two kinds of information, including (*i*) the table entry, by which the debugging platform can distinguish different specified flows, (*ii*) the identifier of the watch snippet, by which the debugging platform can identify the location of the watch snippet. As is shown in Fig. 4, when the operator uses the watch primitive to observe *flow A* and *flow B*. P4DB will install one watch snippet for every MAT. If both flows do exist, then the watch snippets will report the flow traces to the debugging platform. For example, if operators issue 'watch -s 10001 -f {ip_dst:1.1.1.1}', P4DB installs watch snippets for each MAT in the switch whose id is 10001. The match field of the installed watch snippets is the IP destination address (ip_dst).

Break snippet. As is shown in Fig. 5, the break snippet, together with the predication snippet, match snippet, and action snippet enables operators to debug the MAT in a fine-grained way. When an operator issues the break primitive for one MAT, P4DB decomposes the MAT and installs the break snippet. The decomposition of one match-action table enables operators to use next to inspect the changing of PDP states as well packet headers after each step. The break snippet is implemented by one MAT and reports the data plane states, including packet headers, metadata, etc., to the debugging platform when triggered by the specified flow. The triggering of the break snippet will not stop the traffic. Instead, it creates a simulative environment. The break snippet will not stop/pause the real traffic because P4DB cannot control the server injecting the traffic. However, P4DB internally filters the redundancy of the reported data sent from the break snippet due to the continuous triggering of packets. As the break snippet is continually triggered, P4DB will allow operators to use *next* to debug the MAT in a single step.

Predication snippet. The predication snippet residing after the predication step is implemented by one MAT to report processing results of the programmable elements referenced in the *if-else* expression. If the original MAT does not have any predication expression, the predication step will do nothing and pass the flow to the match step. Notably, the specified flow will firstly be matched in the predication step, then be passed to the predication snippet, while the normal flow will not be passed to the predication snippet. P4DB presents predication expressions as well as values of referenced variables.

Match snippet. The match snippet, implemented by one MAT, reports the match fields and match results for the specified flow in the match step. With this snippet, operators can inspect which table entry is hit by the particular flow, and verify the correctness of match rules. The match snippet can be set up to process the target flow.

Action snippet. The action snippet, implemented by one MAT,

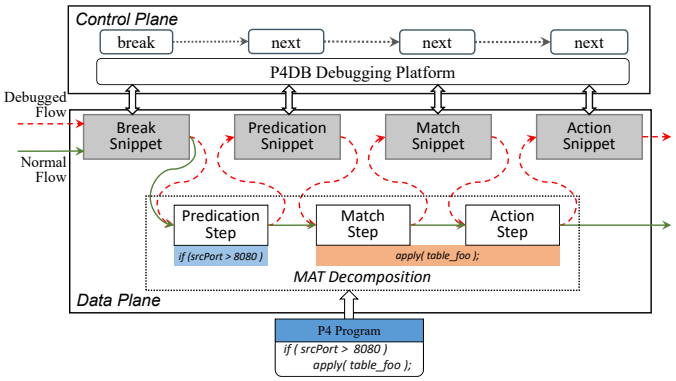


Fig. 5. Design of break snippets, predication snippets, match snippets and action snippets.

reports packet headers, actions and action parameters. By the action snippet, operators can verify whether a P4 program processes a specified flow as expected. Notably, the action step will process the specified flow and pass the flow to the action snippet. Then in the action snippet, the specified flow will trigger the snippet to report which actions and parameters have been taken in the action step. Match rules and actions in action snippets are instantiated when P4DB installs the action snippet. Action snippets reports information including packet headers and variables that are referenced in the actions.

E. An Extension of Debugging Snippets

Debugging snippets presented in §III-D can provide the partial visibility of P4 programs. Moreover, for the P4-specific PDP model, these debugging snippets require dynamic loading or incur moderate PDP resource overheads, which inevitably comes with *limited feasibility*. To overcome the feasibility issue, we design an extension of debugging snippets, namely *pipeline snippet*, which can supply full visibility of PDPs. As is shown in Fig. 6, the pipeline snippet extends previous debugging snippets from two perspectives.

Reporting MAT at the end of the control flow. Debugging snippets can reside at any position of the control flow. Thus, they can concentrate on the part of the control flow. Unlike the previous ones, there is one pipeline snippet per control flow. To avoid updating P4 programs, the pipeline snippet contains a Reporting MAT residing at the end of the control flow and reports packet processing results of all MATs. The packet processing results include metadata and headers that are modified in compound actions. However, the processing results of all MATs might be too large to be reported at once. For example, *Switch.P4* requires over 1000 bytes to contain all the processing results. We can have an observation that for a particular packet, some MATs may not be traversed, and the processing results of these MATs for this packet are not valid. Based on the observation, we can report processing results of a few MATs for a specific packet. Thus, we incorporate multiple compound actions which respectively report processing results for different MAT sets. As is shown in Fig. 6, each compound action of Reporting MAT reports processing results of two MATs, *e.g.,* the first compound action uses *generate_digest* to report processing results of t_1 and t_2. If operators want to

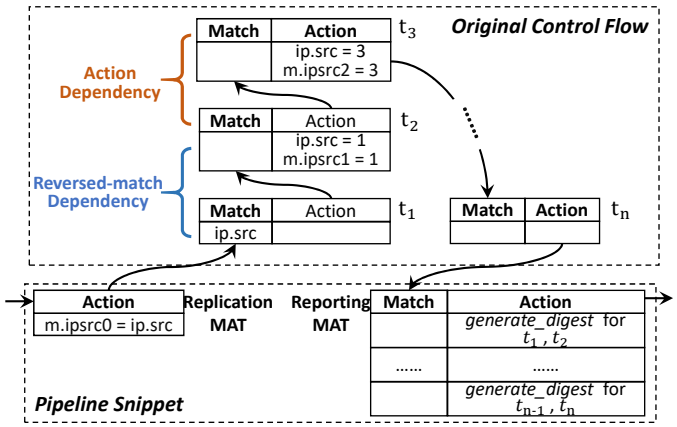


Fig. 6. Design of the pipeline snippet composed of two MATs. The field $ip.src$ triggers reversed-match dependency between t_1 and t_2, and triggers action dependency between t_2 and t_3.

oversee the processing results of other MATs for a specific flow, they can update the corresponding table rules in Report MAT without updating the P4 program.

Field replication in the control flow. Due to the action dependency and the reversed-match dependency [38], processing results of some MATs can be over-written by the subsequent MATs. With the two dependencies, it is hardly possible for the pipeline snippet to attain processing results of all MATs without ambiguity. To overcome issues caused by the dependencies, we replicate fields with specific metadata (*m* in Fig. 6) and label the replicated fields with different versions. Note that we only replicate the fields triggering the action dependency and the reversed-match dependency instead of all fields. Besides, we cannot replicate fields that trigger the reversed-match dependency with the first MAT (*i.e.,* t_1 in the figure), so we should replicate those fields with an additional MAT, namely Replication MAT, at the beginning of the control flow. Then, the pipeline snippet will report the original fields as well as the replicated fields. Thus, we can get the processing results of every MAT in the control flow.

Adding the pipeline snippet and field replication can be automatically completed at the development stage. Compared with the debugging snippets in §III-D, the pipeline snippet does not require frequent updating of P4 programs, which embraces better feasibility. Furthermore, the pipeline snippet supports simulating functionality of other debugging snippets, as the pipeline snippet can supply full visibility of P4 control flow. However, the pipeline snippet inevitably comes with compromises. It can only detect packets that can reach at the pipeline end. In some cases (*e.g.,* wrong match implementation of SmartNIC), packets are dropped in the middle of the control flow, disabling the pipeline snippet. In this respect, the pipeline snippet is not aimed at replacing the previous debugging snippets and can cooperate with them to debug PDPs.

F. Management of Debugging Snippets

P4DB adopts two ways to manage (installation/deletion) the debugging snippets. One is the on-demand way, and the other is the preset way. Both ways have their pros and cons. As for the on-demand way, P4DB will not install the debugging snippet until operators issue the corresponding debugging primitive, and will delete the debugging snippet after operators issue another debugging primitive. For example, P4DB will delete the predication snippet and install the match snippet only after operators issue second next. The break snippet, the predication snippet, the match snippet, and the action snippet are designed in this way. The performance overhead incurred by this way is relatively small, since it does not need to install all relevant debugging snippets into P4 programs.

As for the preset way, P4DB installs all related debugging snippets once the operators issue the debugging primitives. However, only the named debugging snippets are activated while the others are muted. The watch snippet and the pipeline snippet are designed in this way. When operators issue the watch primitive, P4DB will install watch snippets for every MAT in the P4 program. This way may suffer from a performance overhead of multiple debugging snippets running in the data plane. However, this way of installing snippets can collect much more PDP states in case that the specified flow is too short to be consistently debugged.

G. Performance Optimization

In a network, millions of packets may pass through the data plane in every second. P4DB provides operators with an on-the-fly debugging ability by debugging snippets, which inevitably incurs the performance overhead. The purpose of the debugging snippets is to (*i*) filter the specified flow and (*ii*) report the intrinsic states of P4 programs and P4 targets. Accordingly, we propose two designs to reduce the performance overhead in terms of filtering and reporting.

Filter placement. The placement of filtering will directly impact the performance of the data plane. In P4DB, we put forward two ways of placement in consideration of trade-offs for different debugging snippets.

One is to place the filter rules for specified flows outside the debugging snippet, *e.g.*, the filter rules can be placed in the match step, then the match step will filter the specified flow and pass the chosen flow to the match snippet. In this way, only selected flows will be allowed to traverse the debugging snippet, and other normal traffic will bypass the debugging snippet. As is shown in Fig. 5, the predication snippet, the match snippet, and the action snippet adopt this way. This way of placement can efficiently reduce the performance overhead, although it has a limitation in expressing filter rules because it requires that the debugged flow can be exactly matched by the MAT outside the debugging snippet.

The other approach is to place filter rules inside the debugging snippet and use the debugging snippet itself to filter flows. In this way, all traffic, no matter whether it is being debugged, will traverse both the debugging snippet and the original procedure. As shown in Fig. 4 and Fig. 5, the watch snippet, and the break snippet adopt this way of implementation. Although this way may impose an extra overhead for filtering, it offers more flexibility in customizing matching rules for various debugged flows.

Message damper. The reporting traffic directly impacts the performance of the data plane and control channel. A large

volume of reporting traffic inevitably incurs high CPU usage on control planes as well as control channel congestions.

To mitigate the overhead incurred by reporting traffic, we design an data-plane message damper to suppress the reporting traffic. The message damper is implemented via periodically sampling packets of each debugged flow. The message damper maintains an adjustable threshold and a loop counter for each debugged flow. The threshold denotes the period for sampling, while the loop counter counts matched packets in the debugged flow. When the counter reaches the threshold, it will be reset to zero and trigger the debugging snippet to send one reporting packet. The threshold can be dynamically adjusted by operators. The message damper dramatically reduces the performance overhead on the data plane and maintains the debugging functionality of P4DB. If the threshold is too high, the reporting traffic may be too slow to activate the liveness of the breakpoint. Consequently, P4DB will lose the consistent reporting traffic of the bugs, then it will automatically stop the debugging and wait for reactivation of the breakpoint. If the threshold is too low, there will be massive reporting traffic, which impacts the control plane and the control channel.

H. Implementation on Different PDP Models

P4DB hides the heterogeneous implementations of underlying PDP models and provides operators with a unified abstraction of PDP. This generality is internally implemented through maintaining one PDP driver for every type of PDP model. In this paper, we respectively implement P4DB on the P4-specific PDP model and the hypervisor-specific PDP model. Details of implementation can be found in the source code of P4DB. In this section, we will discuss the implementation of decomposed MAT for different PDP models.

In the P4-specific PDP model, the if-else expressions and the MAT are closely tied in the control flow. Therefore, to implement the MAT decomposition, we use two methods. (*i*) For the predication step, we manage to use two MATs to represent the function of the if-else expression equivalently. Based on this technique, the if-else expression can be abstracted and equally expressed by the predication step. (*ii*) For the match step and action step, we add the MAT that merely matches the flow without executing any actions to implement the match step. In the action step, the MAT will execute the actions. As for the hypervisor-specific PDP model, the MAT is already decomposed based on their designs. Therefore, P4DB can be readily applied to the hypervisor-specific PDP model.

IV. Debugging Workflow and Use Cases of P4DB

In this section, we present the workflow of P4DB through a real-world example shown in Fig. 7. Besides, we briefly introduce use cases about applying P4DB to debug runtime bugs on two widely-used P4 targets.

A. Debugging Workflow of P4DB

To demonstrate the workflow of P4DB, we present the procedure of utilizing the *network debugging CLI* to interactively debug a mistakenly-implemented compound action. As shown in Fig. 7, *Host A* is sending packets to *Host C*. However, the

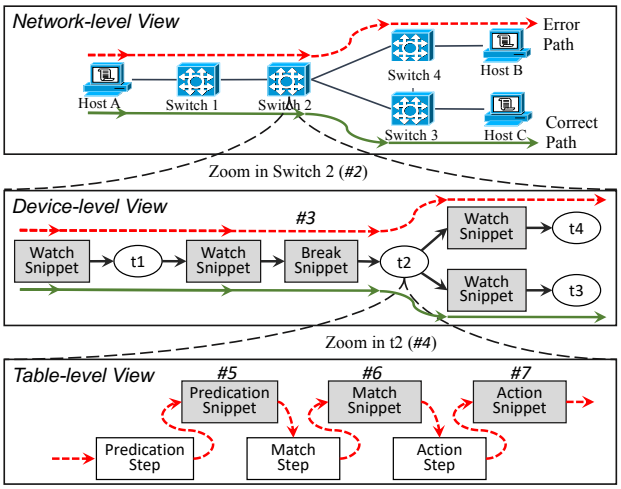


Fig. 7. The hierarchical debugging workflow of P4DB.

packets traverse the error path (the red dashed line) rather than the correct path (the green solid line). Then, operators can employ the CLI to debug the P4 program on *Switch 2*. Next, we will describe the debugging workflow step by step.

#1: The operator initializes the debugging context from the CLI, uses the show primitive to check the name of the PDP instance on *Switch 2*, and issues the attach primitive to attach the debugger to the running instance.

#2: Then the operator starts to debug flows from *Host A* to *Host C* by issuing the watch primitive with parameters of the source address and destination address. Afterward, as long as the specified flow continues, the operator will see the device-level trace of the specified flow.

#3: The operator finds that the trace of the flow is *t1*→*t2*→*t4* rather than *t1*→*t2*→*t3*. Therefore, he decides to debug *t2* in the table-level visibility.

#4: The operator uses the show primitive to get the name of *t2* and issues the break primitive to *t2* with the parameters of the source address and the destination address. Then the consistent flow will trigger the breakpoint.

#5: Afterwards, the operator issues the next primitive to check the predication logic. The CLI provides the original predication expression which is none for *t*2. Nothing is wrong with this step.

#6: Then the operator issues the next primitive again to check the match logic. The operator verifies the match fields and values shown in the CLI and finds nothing wrong either.

#7: The operator issues the third next primitive, verifies the variables and packets referenced in the action step, and finds that one referenced metadata is not modified as expected. Thus this wrong metadata leads to the erroneous branching in the *if-else* statement after *t*2.

#8: The operator checks the primitive actions that are executed in action step, and finally finds that two dependent primitive actions in the compound action are disorderly called. Thus, the bug is found!

#9: Lastly, the operator uses the detach primitive to stop debugging. P4DB removes all debugging snippets in PDP.

B. Two Use Cases of P4DB

In this section, we will introduce our experiences of using P4DB on two widely-used P4 targets. One is BMv2 that is an open-source P4 target for programmers to verify their P4 programs. The other is SmartNIC, a commercial P4-programmable NIC, which has been widely used to offload various network functions [39]. For both cases, the P4 programs are functionally correct but show abnormal behaviors due to the wrong P4 target implementations. Meanwhile, the compiler, as well as the log of these P4 targets, gives no notification of the wrong behaviors. Then, we apply P4DB to debug the P4 targets and locate the elements that are wrongly implemented by those P4 targets. Notably, the three-step MAT decomposition shows its necessity when debugging the runtime bugs incurred by P4 target bugs.

Wrong primitive action implementation on BMv2. As a standard behavioral model, BMv2 is expected to be implemented in complete compliance with the P4 specification. However, we utilize P4DB to find that one of the most important primitive actions, *modify_filed*, is mistakenly implemented in BMv2. Our P4 program is implemented to modify the variable length packet header with *modify_filed*, but we find that packets processed by this program contain many zeros in their headers, which indicates that the packet headers are wrongly modified. Then we attach this program to the CLI to start the debugging procedure shown as follow.

Firstly, we issue watch, but we find that the MAT path was correct. Secondly, we issue break for every MAT and use the break snippet to report the packet headers. Then, we find that when packets traverse the MAT that modifies variable packet-header fields, all bits of the header field are cleared to zero. Thirdly, we decompose this MAT into three steps and watch the packet headers inside the MAT. Then, we find that the predication snippet and the match snippet report regular headers. In the action snippet, the abnormal header field appears, but the reporting messages indicate that the MAT applied the correct compound action with the correct action parameters to the packets. After that, we reason the source code of our P4 program and find that the P4 program invokes *modify_field* to update the packet header field whose length is variable. However, BMv2 handles this invoking abnormally and sets the field to zero without any notification.

Wrong match implementation on SmartNIC SmartNIC is one of the off-the-shelf products that support P4 programming. With the help of P4DB, we find that an oversized ternary match vector can cause black holes in SmartNIC. After being configured with our P4 program, SmartNIC cannot transfer any packets. Firstly, we use watch to trace the MAT path. We find that a MAT discards all packets. Secondly, we install the break snippet in front of the MAT and decompose the MAT. Thirdly, we find that the packets can trigger the predication snippet, but the match snippet does not report any information. After inspecting the match fields of the MAT entries, we find that the packet can match the MAT. The match step could be the source of the bug. Then we cut down the number of ternary match bits in the MAT and reload the program into SmartNIC, and the match snippet can report the packet information correctly.

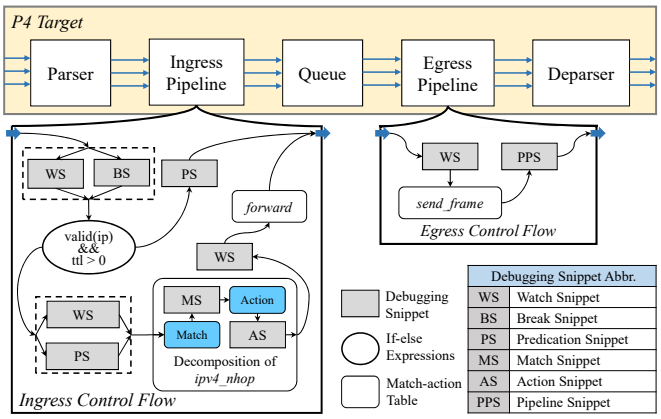


Fig. 8. Router.P4 with debugging snippets.

So we deduce that the oversized ternary match vector causes SmartNIC to drop packets silently, and our further experiment proves that there is a threshold number of the ternary match bits in SmartNIC. If the match vector size of a MAT in a P4 program outnumbers the threshold, SmartNIC will drop all packets that traverse the MAT. The relevant code to verify this bug is published at [40].

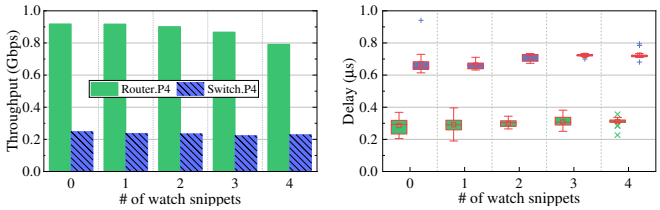
V. Evaluation

A. Overview

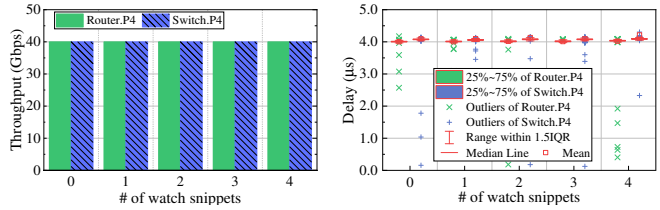
Implementation. We implement P4DB on the P4 controller [10] and evaluate P4DB on the P4-specific PDP model and the hypervisor-specific PDP model. There are three hypervisor-specific PDP models, *i.e.*, MPVisor, HyperV, and Hyper4. We choose MPVisor to conduct our experiments, for its performance improvement and resource efficiency. Besides, to evaluate the performance overhead of P4DB in the real deployment, we implement P4DB on the 3.2T Tofino switch [34] and test the performance overhead incurred by debugging snippets. As MPVisor takes up too much resource which is beyond the constraints of the programmable switch, we only test the P4-specific PDP model on Tofino. Our code is published at https://P4DB.github.io.

Setup. Our experiments are conducted on two off-the-shelf servers, either of which has 2×4 Intel E5-2637 CPU 3.50Ghz cores and 64GB memory. The P4 controller and BMv2 run on different servers. We utilize MoonGen [41] as the packet generator to emit and collect test packets. Moreover, each server is equipped with a dual-port Intel 82599 NIC, so the MoonGen traffic generators running on two servers manages to generate up to 40G traffic. In our evaluation, we utilize *Router.P4* and *Switch.P4* [12] as the P4 programs under testing. As is shown in Fig. 8, *Router.P4* has 3 MATs and 1 if-else expression. *Switch.P4* has as many as 129 MATs. We install six kinds of debugging snippets into the programs.

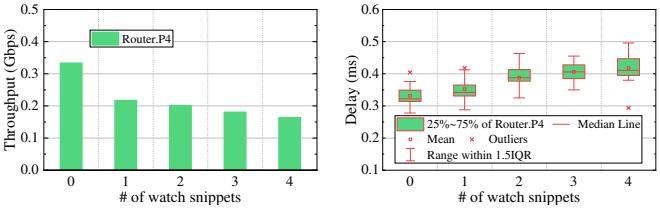
Metrics. First, inserted debugging snippets can influence the throughput and delay of P4 programs, so we conduct experiments regarding the performance of debugging snippets. Second, debugging snippets can generate a large volume of reporting traffic which may congest the control channel and exhaust the CPU resource of the controller. To this end, we conduct a case study on a typical tool based on P4DB, the



(a) Throughput of the P4-specific PDP model on BMv2. (b) Delay of the P4-specific PDP model on BMv2.

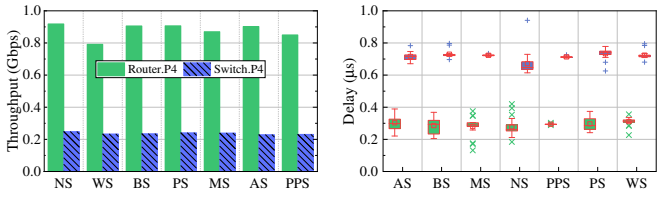


(c) Throughput of the P4-specific PDP model on Tofino. (d) Delay of the P4-specific PDP model on Tofino.

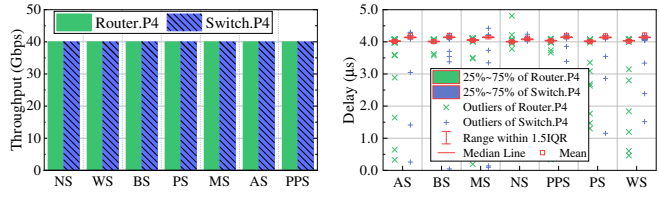


(e) Throughput of the hypervisor-specific PDP model on BMv2. (f) Delay of the hypervisor-specific PDP model on BMv2.

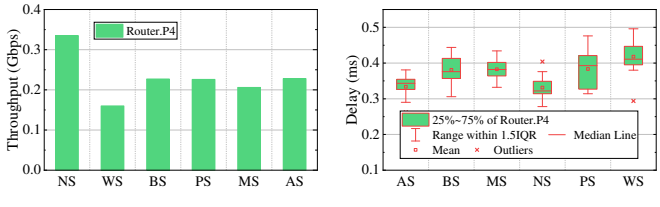
Fig. 9. Performance of watch snippets on two PDP models (*Group 1*).



(a) Throughput of P4-specific PDP model on BMv2. (b) Delay of P4-specific PDP model on BMv2.



(c) Throughput of the P4-specific PDP model on Tofino. (d) Delay of the P4-specific PDP model on Tofino.



(e) Throughput of the hypervisor-specific PDP model on BMv2. (f) Delay of the hypervisor-specific PDP model on BMv2.

Fig. 10. Performance of debugging snippets on two PDP models (*Group 2*).

flow path monitor, to evaluate the reporting message density as well as CPU usage of the controller.

B. Data Plane Performance of P4DB

As for data plane performance, we employ three groups of tests to demonstrate how different determinants can impact performance on BMv2 and Tofino. The three groups are conducted based on the hypervisor-specific PDP model and the P4-specific PDP model respectively. Within each group, we conduct two experiments to evaluate throughput and delay of P4DB. As for throughput, in the experiments, the CPU usage of BMv2 is around 300%. As for the delay, we conduct 100 times of the delay tests and present the inter-quartile (IQR) range to show whether debugging snippets impact delay jitters. For each test, we use the case without debugging snippets as the baseline and run P4DB on two P4 programs, *Router.P4* and *Switch.P4*. We do not implement *Switch.P4* on the hypervisor-based PDP due to the implementation complexity. The metrics for each group are summarized as follows:

Group 1 Performance benchmarks in terms of different number of watch snippets without the message damper.

Group 2 Performance benchmarks in terms of different types of debugging snippets without the message damper.

Group 3 Performance benchmarks in terms of different damper thresholds and different numbers of rules.

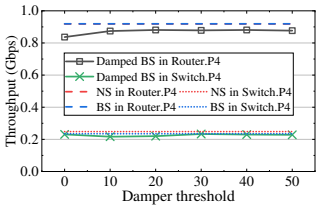
Analysis of Group 1. Fig. 9 shows the performance with different numbers of watch snippets on two PDP models.

Throughput. As is shown in Fig. 9(a) and Fig. 9(e), debugging snippets introduce a moderate throughput degradation on the P4-specific PDP model and the hypervisor-specific PDP model. Moreover, as watch snippets grow in number, the throughput of *Router.P4* descends to a lower extent. For *Switch.P4*, the throughput degradation is much smaller than that of *Router.P4*, as *Switch.P4* has much more MATs than *Router.P4*. Fig. 9(c) shows that watch snippets do not introduce obvious performance overheads on the Tofino switch. However, the hypervisor-specific PDP model has complicated internal control logic and requires tens of physical MATs to implement a logical MATs. Thus, adding a MAT on the hypervisor-specific PDP model equals to adding tens of physical MATs and comes with obvious throughput degradation.

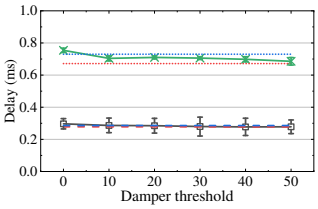
Delay. As shown in Fig. 9(b), the number of watch snippets has a little influence on the P4-specific PDP model running on BMv2. Moreover, delay increase of *Switch.P4* is smaller than that of *Router.P4*. As *Switch.P4* has more MATs than *Router.P4*, the delay of *Switch.P4* is also larger than that of *Router.P4*. Nevertheless, Fig. 9(f) shows the watch snippets incur a moderate delay increase on the hypervisor-specific PDP model. As for Tofino, Fig. 9(d) shows that every snippet only incurs a delay increase of several nanoseconds which account for less than one percent of the total delay.

Analysis of Group 2. Fig. 10 illustrates the performance of P4 programs when embedding different snippets. *NS* which denotes the baseline P4 program. The abbreviations such as *NS*, *MS*, *etc.*, are illustrated by the abbreviation table in Fig. 8. For every case in these experiments, we only install one type of snippets at one time. Besides, in the *WS* case, we install four watch snippets to attain table paths of packets. As the pipeline snippet is dedicated for the P4-specific PDP model, we do not evaluate it on the hypervisor-specific PDP model.

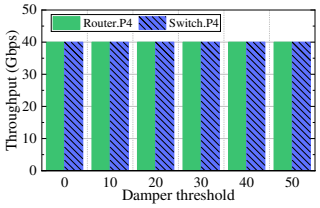
Throughput. Fig. 10(a) and Fig. 10(e) show throughput



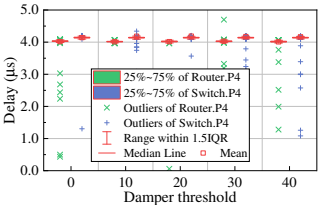
(a) Throughput of the P4-specific PDP model on BMv2.



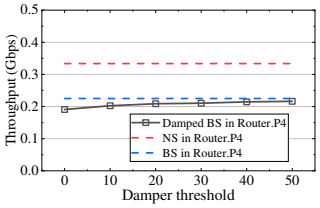
(b) Delay of the P4-specific PDP model on BMv2.



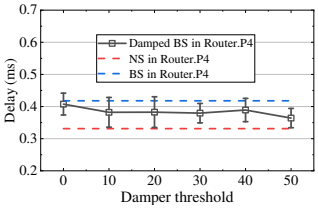
(c) Throughput of the P4-specific PDP model on Tofino.



(d) Delay of the P4-specific PDP model on Tofino.



(e) Throughput of the hypervisor-specific PDP model on BMv2.



(f) Delay of the hypervisor-specific PDP model on BMv2.

Fig. 11. Performance of the break snippet with the message damper on two PDP models (*Group 3*).

with different types of debugging snippets on BMv2. Notably, the *WS* case performs worst because it needs to embed four watch snippets into P4 programs. Comparing with the baseline, it merely achieves 791.6 Mbps with a throughput penalty of 13.8% on the P4-specific PDP, and 163.7 Mbps with a throughput penalty of 51% on the hypervisor-specific PDP. However, on the P4-specific PDP model, other types of debugging snippets only have a performance degradation of few percents. Besides, the throughput penalty on MPVisor is larger than on the P4-specific PDP model and ranges from 32.7% to 38.5% when being compared with the baseline. As for the throughput results shown in Fig. 10(c), the debugging snippets do not introduce any throughput penalty on the programmable switch. For all debugging snippets, Tofino keeps forwarding packets at 40 Gbps. Besides, for *Switch.P4*, all debugging snippets only incur little throughput degradation.

Delay. As is shown in Fig. 10(b) and 10(f), results differ among two PDP models. As for the P4-specific PDP model, there is little difference between the baseline and various debugging snippets in delay. However, for MPVisor, the increase of the delay ranges from 15% to 26% when being compared with the baseline. As shown in Fig. 10(d), the debugging snippets almost introduce no delay increase on Tofino.

Analysis of Group 3. For this group, we test the performance of debugging snippets with two metrics, *i.e.,* damper thresholds when the message damper is enabled and installed rules in debugging snippets. (1) As for damper thresholds, throughput and delay will be measured with different damper thresholds. Besides, as there is little difference between the damped break snippet and the two counterparts, we adopt the bar chart to show the throughput and the box-plot to show the delay. (2) As for rules in debugging snippets, we evaluate the performance of *Router.P4* with a BS on Tofino, and the number of rules in the BS increases from 10 to 100K.

Damper threshold. As is shown in Fig. 11(a) and Fig. 11(e). The throughput increases with the threshold. Even if the threshold reaches 50, the throughput is still smaller than the throughput of BS without a damper, because the message damper itself incurs performance overheads. Besides, on the P4-specific PDP model, when the threshold is larger than 10, the increasing of throughput tends to be stable. On the hypervisor-specific model, the increasing of throughput becomes stable after the threshold is larger than 30. As shown in Fig. 11(c), the damper threshold has no impact on the throughput on Tofino which can forward packets at 40 Gbps for all tested damper thresholds. Fig. 11(b) and Fig. 11(f) show that the damper has a positive effect on delay. With the damper threshold increasing, the delay deceases. On the P4-specific PDP model, the delay can be the same as the baseline delay. Fig. 11(d) shows the packet-processing delay on Tofino, the damper threshold does not influence delay.

# of rules. We respectively show throughput and delay of *Router.P4* with varied numbers of rules in BS, which is shown in Fig. 12. Furthermore, as one stage of Tofino could not hold all rules. Thus, BS could occupy multiple stages to store all corresponding rules. We can summarize from Figure 12 that the number of rules installed in BS does not bring obvious performance degradation on Tofino.

Summary. The performance of P4DB is different on the P4-specific PDP model and the hypervisor-specific PDP model. However, the difference can be mainly attributed to the performance degradation caused by MPVisor itself. Although the hypervisor-specific PDP model provides dynamic PDP reconfiguration, it suffers from the performance overhead of model translation. For the P4-specific PDP model, P4DB incurs a minor performance overhead but faces interruption of the PDP, even though the interruption time is usually under 50ms [34]. Notably, on Tofino, debugging snippets almost introduce no performance overhead. P4DB makes a trade-off between *runtime visibility* and *performance*. Furthermore, we argue that introducing the promising ability of on-the-fly debugging for PDP with acceptable performance degradation is worthy, especially in the case of network outages.

C. Control Channel and Control Plane of P4DB

In this section, we use the performance of the *flow path monitor* running on P4DB as a case study to explore the feasibility and ability of P4DB. Besides, there remains a concern that the messages generated by debugging snippets could overwhelm P4DB running on the centralized controller, which inevitably limits the scalability of P4DB. We provide the message damper to mitigate this concern. To this end, we accordingly evaluate (*i*) debugging messages in the control channel and (*ii*) the CPU usage of the P4DB, when the *flow path monitor* is activated.

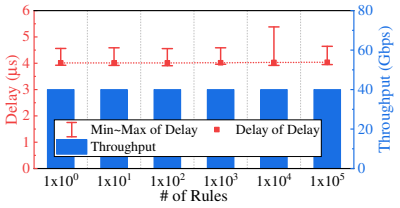


Fig. 12. Performance of Router.P4 on Tofino with different numbers of rules.

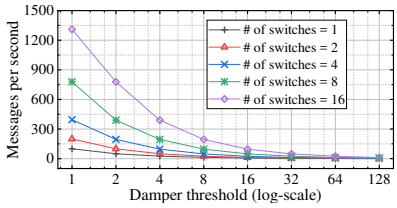


Fig. 13. Debugging messages per second in the control channel.

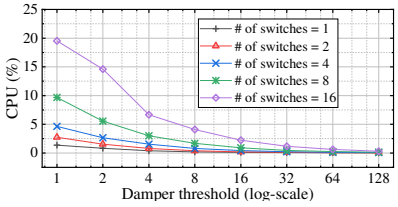


Fig. 14. CPU usage of the flow path monitor with different damper thresholds.

The *flow path monitor* keeps track of forwarding paths for packets of designated flows. Thus, the monitor needs to attain the ingress port and egress port of the packets on each hop. So the hop number can directly influence the workload of the monitor. Besides, the message damper could significantly cut down the workload, so we can also use the damper threshold as another variable. Furthermore, we utilize the messages (per second) in the control channel and the controller CPU usage to illustrate the performance of the monitor. For the evaluation, we configure the monitor to keep track of a flow which emits one hundred packets per second. With the number of the traversed switches and the damper threshold varying, we can have the following analysis.

Analysis of debugging messages. For this experiment, the number of the messages is collected at the centralized P4DB platform. As shown in Fig. 13, we have to admit that the overall workload imposed by the monitor is not negligible, which potentially influences the scalability of P4DB. However, through the message damper, P4DB can considerably cut down the number of messages in the control channel without compromising the functionality of the *flow path monitor*. When the damper threshold reaches at 128, *i.e.,* the watch snippet report a message per 128 packets, there are only several messages per second in the channel, even if packets of the flow need to traverse sixteen switches.

Analysis of CPU usage. The CPU usage of the centralized controller is measured to illustrate two issues. The first is the scalability issue that may be caused by P4DB. As can be seen in Fig. 14, when the damper threshold is set to one, the *flow path monitor* requires more CPU resource to process the workloads as the number of traversed switches increases. The second one is the impact of the message damper. We can see that when the message damper is added into the debugging snippet, the CPU usage declines rapidly. Moreover, the CPU usage declines to below 0.6% in all tested scenarios, when the damper threshold is set to 64.

Summary. P4DB imposes a certain overhead on the control channel and the control plane. Although the performance overhead is primarily determined by the volume of the traffic that is being debugged on the data plane, operators can adaptively adjust the damper threshold to reduce the overhead from the reporting traffic generated by debugging snippets. Overall, our experiments show that the messages damper performs powerfully in terms of improving the scalability of P4DB.

VI. Related Work

P4 verification has attracted a lot of attentions. P4 verification tools [27]–[29], [42] convert P4 programs to existing models and employ mature techniques to identify bugs in P4 programs and table rules. For example, in [25], the authors proposed a static analysis tool that compiles P4 to Datalog. Then, the verification model can be automatically updated as the P4 program changes. However, they cannot diagnose bugs that happen in P4 targets, such as faulty P4 target implementations. P4DB provides operators with full visibility of the data plane states at runtime and can handle various types of runtime bugs.

Network troubleshooting has long been an important topic which attracts a lot of researching efforts. Passively tracking how packets are processed inside switches is a general approach for network troubleshooting. Based on this approach, a line of tools have been designed. (1) ndb [13] and NetSight [14] track all packets on data planes and provide a series of interactive debugging commands to troubleshoot network faults. But they have to generate a huge amount of tracking workloads (*i.e.,* reporting messages), which constrains their scalability. (2) EverFlow [15] relies on the 'match-and-mirror' design to enable tracking matched flows, which reduces the tracking workloads. P4DB also employs match-action tables to track matched flows. Furthermore, P4DB further reduces tracking workloads via message dampers.

VII. Discussion

As previously mentioned, P4DB implements the management of the break, predication, match, and action snippets in an on-demand way, and using the latest inbound traffic as the trigger. Thus, it requires packets to consistently trigger the runtime bugs. We consider this prerequisite of recurrence of runtime bugs is somewhat common. For example, according to [43], troubleshooting network bugs usually lasts for 30-60 minutes which provides plenty of time for operators to troubleshoot the bugs. Besides, for flows that are too short to be debugged, P4DB can collaborate with the event-driven debugging tools and some log-based verification designs to bridge the gap. For example, P4DB can record the reporting messages and perform the verification on the message history, just like NetSight [14]. With the message history, P4DB can timely troubleshoot the runtime errors triggered by short flows.

VIII. Conclusion

This paper is devoted to on-the-fly debugging of runtime bugs for programmable data planes. We proposed P4DB as a general debugging platform that empowers operators to debug various runtime bugs with simplicity and visibility. We proposed three novel designs including (1) the three-step MAT decomposition, (2) debugging primitives, and (3) debugging snippets. Besides, we introduce two methods to optimize the

performance of P4DB and implement P4DB based on the P4-specific PDP model and the hypervisor-specific PDP model respectively. We have built a prototype of P4DB, published the source code and evaluated the prototype in terms of data plane, control plane and control channel. Evaluation results show that P4DB enables operators to conveniently troubleshoot runtime bugs in PDP-enabled networks and only incurs a moderate performance overhead. As for the real deployment of P4DB on the off-the-shelf programmable switch, debugging snippets almost introduce no performance penalty.

REFERENCES

[1] C. Zhang, J. Bi, Y. Zhou, J. Wu, B. Liu, Z. Li, A. B. Dogar, and Y. Wang, "P4db: On-the-fly debugging of the programmable data plane," in *Proceedings of ICNP*, 2017, pp. 1–10.

[2] 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.

[3] H. Song, "Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane," in *Proceedings of HotSDN*. New York, NY, USA: ACM, 2013, pp. 127–132.

[4] 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 SIGCOMM*. New York, NY, USA: ACM, 2017, pp. 15–28.

[5] M. Ghasemi, T. Benson, and J. Rexford, "Dapper: Data plane performance diagnosis of tcp," in *Proceedings of SOSR*. New York, NY, USA: ACM, 2017, pp. 61–74.

[6] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of SOSP*. New York, NY, USA: ACM, 2017, pp. 121–136.

[7] N. Katta, M. Hira, C. Kim, A. Sivaraman, and J. Rexford, "Hula: Scalable load balancing using programmable data planes," in *Proceedings of SOSR*. New York, NY, USA: ACM, 2016, pp. 10:1–10:12.

[8] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé, "Netpaxos: Consensus at network speed," in *Proceedings of SOSR*. New York, NY, USA: ACM, 2015, pp. 5:1–5:7.

[9] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *Proceedings of NSDI*, Renton, WA, 2018, pp. 35–49.

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

[11] ONOS, "P4 support via bmv2 and p4runtime," Website, https://wiki.onosproject.org/display/ONOS/P4+support+via+BMv2+and+P4Runtime.

[12] The P4 Language Consortium, "Consolidated switch repo (api, sai and nettlink)," Website, https://github.com/p4lang/switch.

[13] N. Handigol, B. Heller, V. Jeyakumar, D. Maziéres, and N. McKeown, "Where is the debugger for my software-defined network?" in *Proceedings of HotSDN*. New York, NY, USA: ACM, 2012, pp. 55–60.

[14] N. Handigol, B. Heller, V. Jeyakumar, D. Mazières, and N. McKeown, "I know what your packet did last hop: Using packet histories to troubleshoot networks," in *Proceedings of NSDI*. Seattle, WA: USENIX Association, 2014, pp. 71–85.

[15] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, and H. Zheng, "Packet-level telemetry in large datacenter networks," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 479–491, Aug. 2015.

[16] R. Durairajan, J. Sommers, and P. Barford, "Controller-agnostic sdn debugging," in *Proceedings of CoNEXT*. New York, NY, USA: ACM, 2014, pp. 227–234.

[17] P. Zhang, H. Li, C. Hu, L. Hu, L. Xiong, R. Wang, and Y. Zhang, "Mind the gap: Monitoring the control-data plane consistency in software defined networks," in *Proceedings of CoNEXT*. NY, USA: ACM, 2016, pp. 19–33.

[18] Q. Zhi and W. Xu, "Med: The monitor-emulator-debugger for software-defined networks," in *Proceedings of INFOCOMM*, April 2016, pp. 1–9.

[19] A. Wundsam, D. Levin, S. Seetharaman, and A. Feldmann, "Ofrewind: Enabling record and replay troubleshooting for networks," in *Proceedings of USENIX ATC*. Berkeley, CA, USA: USENIX Association, 2011, pp. 29–29.

[20] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "Automatic test packet generation," *IEEE/ACM Trans. Netw.*, vol. 22, no. 2, pp. 554–566, Apr. 2014.

[21] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proceedings of SIGCOMM*. New York, NY, USA: ACM, 2011, pp. 290–301.

[22] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proceedings of NSDI*. San Jose, CA: USENIX, 2012, pp. 113–126.

[23] K. Bu, X. Wen, B. Yang, Y. Chen, L. E. Li, and X. Chen, "Is every flow on the right track?: Inspect sdn forwarding with rulescope," in *Proceedings of INFOCOM*, April 2016, pp. 1–9.

[24] P. Perešíni, M. Kuźniar, and D. Kostić, "Monocle: Dynamic, fine-grained data plane monitoring," in *Proceedings of CoNEXT*. New York, NY, USA: ACM, 2015, pp. 32:1–32:13.

[25] N. Mckeown, T. Dan, G. Varghese, N. Lopes, N. Bjorner, and A. Rybalchenko, "Automatically verifying reachability and well-formedness in p4 networks," 2016.

[26] A. Nötzli, J. Khan, A. Fingerhut, C. Barrett, and P. Athanas, "P4pktgen: Automated test case generation for p4 programs," in *Proceedings of SOSR*. New York, NY, USA: ACM, 2018, pp. 5:1–5:7.

[27] L. Freire, M. Neves, L. Leal, K. Levchenko, A. Schaeffer-Filho, and M. Barcellos, "Uncovering bugs in p4 programs with assertion-based verification," in *Proceedings of SOSR*. New York, NY, USA: ACM, 2018, pp. 4:1–4:7.

[28] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, "p4v: Practical verification for programmable data planes," in *Proceedings of SIGCOMM*. Budapest, Hungary: ACM, 2018, pp. 1–14.

[29] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging p4 programs with vera," in *Proceedings of SIGCOMM*. Budapest, Hungary: ACM, 2018, pp. 1–14.

[30] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[31] GNU, "Gdb: The gnu project debugger." Website, http://www.gnu.org/software/gdb/.

[32] 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 SIGCOMM*. New York, NY, USA: ACM, 2013, pp. 99–110.

[33] 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 SIGCOMM*. New York, NY, USA: ACM, 2017, pp. 1–14.

[34] Barefoot Networks, "Barefoot tofino switch," Website, https://barefootnetworks.com/technology/.

[35] D. Hancock and J. Van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *Proceedings of CoNEXT*. New York, NY, USA: ACM, 2016, pp. 35–49.

[36] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "Mpvisor: A modular programmable data plane hypervisor," in *Proceedings of SOSR*. New York, NY, USA: ACM, 2017, pp. 179–180.

[37] C. Zhang, J. Bi, Y. Zhou, A. Basit, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *Proceedings of ICCCN*, 2017, pp. 1–9.

[38] L. Jose, L. Yan, G. Varghese, and N. Mckeown, "Compiling packet programs to reconfigurable switches," in *Proceedings of NSDI*. USENIX Association, 2015, pp. 103–115.

[39] Netronome., "Agilio cx 2x10gbe," Website, https://www.netronome.com/products/agilio-cx/.

[40] "Over-sized ternary match cause black holes in smartnic," Website, https://github.com/p4db/p4db-issues.

[41] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proceedings of IMC*. New York, NY, USA: ACM, 2015, pp. 275–287.

[42] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proceedings of NSDI*. Oakland, CA: USENIX Association, 2015, pp. 499–512.

[43] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown, "A survey on network troubleshooting," Website, http://yuba.stanford.edu/~peyman/docs/atpg-survey.pdf.

Yu Zhou Yu Zhou received the B.S. degree from the School of Information and Communication Engineering, Beijing University of Posts and Telecommunications, Beijing, China, in 2016. He is currently pursuing the Ph.D. degree with the Institute for Network Sciences and Cyberspace, Tsinghua University. His research interests include software-defined networking and programmable data planes.

Jun Bi (S'98–A'99–M'00–SM'14) received B.S., C.S., and Ph.D. degrees in Department of Computer Science at Tsinghua University, Beijing, China. Currently, he is a Changjiang Scholar Distinguished Professor of Tsinghua University and the director of Network Architecture Research Division, Institute for Network Sciences and Cyberspace at Tsinghua University. His current research interests include Internet Architecture, SDN/NFV, and Network Security. He successfully led tens of research projects, published more than 200 research papers and 20 Internet RFCs or drafts, owned 30 innovation patents, received national science and technology advancement prizes, IEEE ICCCN outstanding leadership award, and best paper awards.

Cheng Zhang received currently pursuing the Ph.D. degree with the Department of Computer Science, Tsinghua University, Beijing, China. He has published papers in SIGCOMM, ICNP, SOSR, ICCCN, and ISCC. His research interests include Internet architecture, software-defined networking, data plane virtualization, and the programmable data plane.

Bingyang Liu received a B.S. degree in computer software from Tsinghua University, China. He was a joint Ph.D. student in the Department of Computer Science, Duke University. He received a Ph.D. degree in computer science from Tsinghua University, China. His research fields include Internet architecture, DDoS defense, and software-defined networking (SDN).

Zhaogeng Li received the B.S. and Ph.D. degrees from Tsinghua University. He is currently a Senior Engineer with Baidu Inc. His main research interest includes datacenter network, RDMA, information-centric network, and edge computing.

Yangyang Wang received his B.S. degree in computer science and technology from Shandong University, China in 2002, M.S. degree from Capital Normal University, China in 2005, and Ph.D. degree from the Department of Computer Science of Tsinghua University, China in 2013. He is currently a postdoctoral scholar in computer science at Tsinghua University. His research interests include Internet routing architecture, future Internet design, and SDN

Mingli Yu received a B.S degree in computer science and engineering from Tsinghua University, China. He is currently a master student in the Department of Computer Science and Engineering, Pennsylvania State University. His research fields include network reconnaissance and software-defined networking(SDN).