P4DB: On-the-fly Debugging of the Programmable Data Plane

Cheng Zhang*†‡, Jun Bi*†‡, Yu Zhou*†‡, Jianping Wu*†‡, Bingyang Liu §,
Zhaogeng Li *†‡, Abdul Basit Dogar*†‡, Yangyang Wang *†‡
*Institute for Network Sciences and Cyberspace, Tsinghua University
†Department of Computer Science, Tsinghua University
‡Tsinghua National Laboratory for Information Science and Technology (TNList).
§Huawei Technologies Co., Ltd.

***Abstract*—While extending network programmability to a larger degree, P4 also raises the risks of incurring runtime bugs after the deployment of P4 programs. These runtime bugs, if not handled promptly and properly, can ruin the functionality and performance of networks. Unfortunately, the absence of runtime debuggers makes troubleshooting of P4 program bugs challenging and intricate for operators. This paper is devoted to the on-the-fly debugging of runtime bugs in P4-enabled networks. We propose P4DB, a general debugging platform that empowers operators to debug P4 programs in three levels of visibility by provisioning operator-friendly primitives. By P4DB, operators can use the *watch* primitive to quickly narrow the debugging scope from network level or device level to table level, then use the *break* and *next* primitives to decompose the match-action table into three steps and troubleshoot the runtime bugs step by step. We implemented a prototype of P4DB and evaluated the performance in terms of the data plane, control plane and control channel. On P4-specific programmable data plane, P4DB merely introduces a small throughput penalty (1.3%∼13.8%) and imposes a little-increased delay (0.6%∼11.9%).**

I. Introduction

P4 [1], a recently proposed domain-specific language, enables operators to customize the behavior of a programmable data plane (PDP). It allows operators to make extensive innovations on the network protocols by decoupling custom protocol implementations from underlying switch code.

To support new protocol formats and operations, P4 empowers operators to define various programmable elements in a P4 program. Operators can customize the `parser` to extract header fields complied with particular protocol formats. In the match-action table (MAT), operators can define the `match fields`, the permissible `compound actions` and `primitive actions`. Moreover, operators can organize various MATs as a complex Direct Acyclic Graph (DAG) in the `control flow`. Besides, operators can declare data plane variables such as `metadata`, `registers`, to perform complex protocol operations. At runtime (i.e., while the switch is forwarding packets), the controller can manage the `table entries` in the MATs. Many, recently, proposed researches, such as [2], [3], show that P4 can greatly accelerate the innovation of network protocols and functions.

However, with so many programmable elements being available for manipulation, the P4 programs, like all other software developed by human, are inevitably prone to errors. These errors can be mainly categorized into two types. One is compile-time errors that can be detected at compile time. Another is runtime errors that cannot be detected at compile time and may cause diverse misbehaviors after the program is deployed onto the device. In this paper, we are the first to focus on debugging the runtime bugs of P4 programs.

978-1-5090-6501-1/17/$31.00

Debugging the runtime bugs of P4 programs is a rather difficult task, and faces the following challenges:

Diversity. Runtime bugs may happen to any programmable elements in diverse styles. For example, operators may mistake a wrong `metadata` for the right one when defining the predication expression (i.e., *if-else* statement), make a logic error in defining the `control flow`, etc. Accordingly, these bugs can cause diverse kinds of misbehaviors at runtime such as malformed packets, packet loss, packet loops, etc.

Complexity. As P4 programs grow in size, operators can be overwhelmed by the complexity of troubleshooting runtime bugs. Take the *switch.p4* program in the P4 github repository [4] as an example. The *switch.p4* contains over 10K LoC, 129 MATs, 76 predication expressions and 340 `compound actions`. Thus, when an operator wants to find out why a packet is not forwarded as expected, he has to dump all table entries in MATs and reason the bugs out table by table. The dynamic `metadata` referenced in *switch.p4* further adds up the debugging complexity.

Invisibility. In the P4 program, most programmable elements such as `metadata`, `control flow`, `primitive action`, etc., are not visible to operators at runtime, but are vital for debugging. For example, once the program is deployed onto the device, the logic defined by the `control flow`, like "hard-coded logic", cannot be observed by operators. Thus, operators cannot directly see the MAT path (trace of the packet among different MATs) of the packet. This invisibility of programmable elements necessarily aggravates the difficulty in debugging programs at runtime.

Therefore, with the booming of the PDP, debugging the runtime bugs is becoming an intricate and challenging task. However, existing debugging tools are mainly dedicated for OpenFlow-based [5] SDN and cannot be simply ported to handle runtime bugs of P4-enabled networks. Some of them are designed for the control plane and are out of the scope of this paper for their inability of verifying data plane behaviors. The others are designed for the data plane and can be categorized into two types:

Firstly, debugging tools such as [6], [7], [8], [9], [10], [11], etc., are based on the runtime-monitoring technique. They merely focus on the `table entries` without consideration

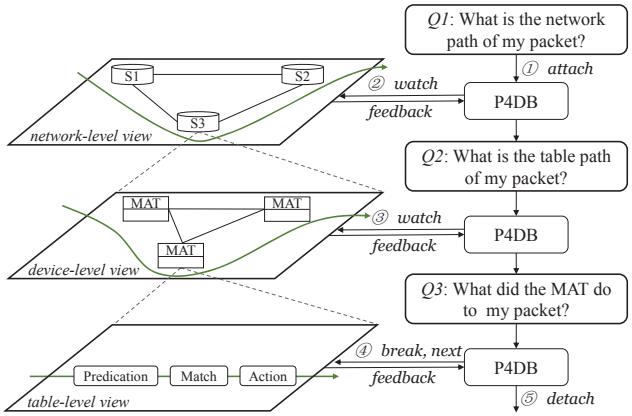


Figure 1. Debug the PDP program with P4DB

of other programmable elements. Besides, they also suffer from the large performance overhead of monitoring.

Secondly, other debugging tools such as [12], [13], [14], [15], [16], are based on the static analysis technique. Due to the limitation of static analysis, they cannot adaptively model the changes to forwarding behaviors without reprogramming the internals. Moreover, they usually regard the data plane model as the correct input model to generate the probe packets or simulation baseline. Thus, they cannot detect the bugs inside the P4 program and will cause false-positive problems.

Debugging the PDP is a rather new topic. As far as we know, there are two debugging tools for P4. However, neither of them can be utilized to troubleshoot the runtime bugs. A static analysis tool is proposed in [17] to verify the reachability and well-formedness of packets. However, this tool cannot be used to troubleshoot the runtime bugs due to the limitation of static analysis. Besides, the software debugger in BMv2 project is used to find the bugs in the development stage. However, guaranteeing the correctness in the simulation environment cannot ensure the correctness after deployment.

This paper introduces P4DB, a general debugging platform with three novel designs, to address aforementioned challenges.

1) The **debugging primitives** provide simple and usable interfaces for operators to simplify the debugging workflow (***Complexity***). So operators can avoid dumping all table entries in MATs and reasoning the bugs out table by table. These debugging primitives include attach, detach, watch, break, next, etc.

2) The **debugging snippets** are designed to report states of diverse programmable elements on data plane (***Diversity*** & ***Invisibility***) and implement debugging primitives. A debugging snippet, in essence, is an on-data-plane code piece that can be flexibly inserted into special positions within the P4 program and report user-interested states to P4DB at runtime.

3) The **three-step decomposition** is devised to decompose the MAT along with its predication expression equivalently into three steps including predication step, match step and action step (***Diversity*** & ***Invisibility***). Therefore, operators can use break and next primitives to inspect in detail the execution of the MAT in a single-step way.

By P4DB, operators can flexibly scrutinize the P4 program in three different 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 see the processing of the packet step by step in the decomposed MAT.

Therefore, as shown in Figure 1, when debugging the P4 program, operators can use the watch primitive to narrow quickly the debugging scope from network level to some specific table level. Then operators can use the break and next primitives to decompose one MAT and to observe in detail what is happening to the packet at runtime.

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 used to troubleshoot runtime bugs in P4-enabled networks. We demonstrate P4DB's viability and usability by a real-world example.
- In P4DB, we propose three novel designs to (*i*) facilitate troubleshooting various runtime bugs, (*ii*) simplify the debugging workflow and (*iii*) provide different levels of visibility.
- We implemented a prototype of P4DB based on two recently proposed PDPs: P4-specific PDP and hypervisor-specific PDP. Accordingly, we made comprehensive evaluations of P4DB. Results indicate that P4DB merely takes a small performance overhead.

The remainder of this paper is organized as follows. We discuss the related work in section II. Then Section III describes the philosophy and generality of P4DB. The key designs of P4DB are shown in Section IV. Afterwards, in Section V, the debugging workflow of P4DB is demonstrated through a real-world example. In Section VI, we evaluate the performance of P4DB. Finally, we discuss the feasibility of P4DB in Section VII, and make a conclusion in Section VIII.

II. Related Work

At the time of writing this paper, few researches focus on debugging the P4 programs. In a technique report [17], the author proposes a static analysis tool that can compile P4 to Datalog, so that this verification model can be automatically updated as the P4 program is changed. However, due to the constraint of static analysis, this tool cannot handle the runtime bugs. Besides, as mentioned above, this tool also takes P4 programs as the input, and cannot resolve the false-positive problems. On the contrary, P4DB provides operators with full visibility of the data plane states at runtime and is not designed for any specific type of runtime bugs.

The software debugger provided in BMv2 [4] enables operators to debug the P4 program during development stage. However, this tool only provides a coarse-granularity verification at the development stage, and is incapable of handling runtime bugs after deployment.

As previously discussed, other debugging tools are proposed to debug the data plane in OpenFlow-based SDN, and are not

feasible for P4-enabled networks. Due to the space reason, we only discuss the most relevant researches as follows.

In particular, ndb [6] and its successor NetSight [7] seem to share a similar idea of providing interactive debugging commands with P4DB. For example, ndb provides the *breakpoint* primitive. However, according to the author, the *breakpoint*, in essence, is a trace point which is similar with our watch primitive. The reason for keeping the breakpoint terminology is to maintain its familiarity in program debugging. Contrarily, the break primitive in P4DB focuses on the behavior of the MAT instead of packets, and innovatively decomposes the MAT into three sequential steps. Besides, P4DB also enables operators to use next primitive to debug the MAT in a single-step way after the break snippet is triggered.

Moreover, even taking the traffic compression technique used in ndb and NetSight into account, they still suffer from a large overhead of generating postcard traffic for every packet on the data plane. Additionally, they need to modify the data plane to implement the debugger. Comparing with ndb and NetSight, P4DB utilizes the on-data-plane damper to suppress efficiently the reporting traffic, and does not need to modify the implementation of the PDP.

III. Overview of P4DB

A. The Philosophy of P4DB

Most bugs arise from mistakes and errors made in either a program's source code or its design. Therefore, the most efficient way to debug is to allow programmers to locate directly and see what is happening. In operating systems, programmers can load the application's source code into the debugger such as the gdb, then insert the breakpoint into the source code and directly check what is happening at runtime. When a programmer sets the breakpoint for the program. Internally, a small piece of "trap code", that switches the execution subject from the program to the debugger, is dynamically inserted into the program by gdb.

P4DB shares the similar basic idea with gdb in operating systems. Although, there is no trap instruction that enables operators to stop the execution of P4 programs, in P4DB, pieces of "report code", called debugging snippets, are designed to report the states of the programmable elements. Actually, a debugging snippet is a simple DAG of MATs devised to report the on-data-plane states of user-interested flows. Besides, different combinations of debugging snippets can be used to implement the user-friendly debugging primitives for operators. So operators can load the P4 program, use debugging primitives to embed dynamically the debugging snippets into 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 can be based on various match-action-table architectures including RMT [18] and dRMT [19].

B. The Generality of P4DB

Firstly, we will illustrate two recently proposed PDPs as well as their pros and cons. Secondly, we will describe the features that can be utilized by P4DB based on various PDPs.

1) The P4-specific PDP: The P4-specific PDP is the most widely adopted and a pioneer in defining the PDP. In the P4-specific PDP, the language model used in the P4 program, is closely tied to the implementation model on the hardware device. Therefore, P4 programs can fully benefit from the high performance of the hardware device without any overhead of model translation [20]. However, this tight coupling between the high-level program and the low-level hardware device can also lead to the result that every time operators change the P4 program, the interruption and reconfiguration of the hardware device are unavoidable.

2) The Hypervisor-specific PDP: Recently, hypervisor-specific PDPs such as Hyper4 [21] and MPVisor [22], are proposed to decouple the high-level P4 program and the low-level hardware device by adding a layer of light-weight hypervisor in-between. In this way, although there is a performance overhead due to the additional hypervisor, P4 programs can be changed without interrupting hardware devices.

3) Features on Various PDPs: Both the P4-specific PDP and hypervisor-specific PDP have their own pros and cons, and present their unique features in addition to the common programmability of the programmable elements.

Therefore, P4DB is designed as a general debugging platform that presents *general debugging features* by utilizing the common programmability provided by various PDPs. Meanwhile, P4DB also presents the unique features of various PDPs as *special debugging features* when being used to debug the specific PDP. For instance, based on the hypervisor-specific PDP, P4DB can benefit from the uninterrupted feature, while it suffers from a performance overhead of model translation. Oppositely, based on P4-specific PDP, P4DB provides high performance and the language compatibility, but faces interruption of the data plane. The details of implementation upon various PDPs will be described in IV-F.

IV. Design of P4DB

A. System Architecture

Figure 2 shows the architecture of P4DB. The automatic debugging tool and command line interface (CLI) debugging tool can be developed based on the debugging primitives provided by the debugging platform. In this paper, we will illustrate the default CLI debugging tool in P4DB. The debugging primitives, internally, are implemented by different compositions of debugging snippets. The PDP program manager maintains the status of all P4 programs as well as their running instances. Reconfiguration of the device will trigger the PDP program manager to recompile the corresponding P4 program. The PDP model manager maintains mapping of the particular PDP model and the corresponding hardware device. The debugging message service maintains the control channel between on-data-plane debugging snippets and the debugging platform.

B. Three-step Decomposition of the MAT

In P4DB, we decompose one MAT into three sequential steps: predication step, match step and action step. Predication step (implemented by two MATs) is functionally equivalent to the *if-else* statement bounded with the MAT in the `control`

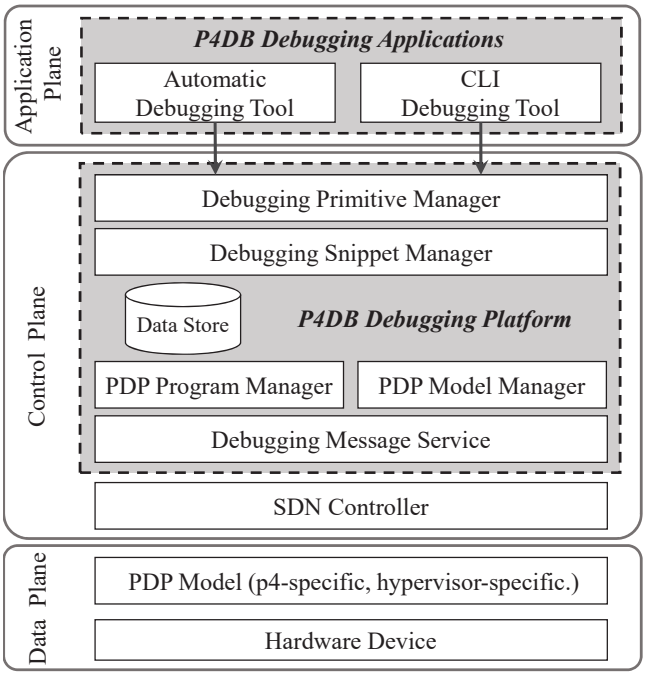


Figure 2. Architecture of P4DB.

`flow`. Similarly, the match step (implemented by one MAT) will only execute the match logic of the original MAT without executing any actions. The action step will merely do the action logic.

Based on this decomposition, P4DB can insert predication snippet, match snippet and action snippet after each corresponding decomposed step; collect the data plane states from the debugging snippet; respectively show data plane states for each step. Accordingly, operators can orderly use three next primitive to verify the correctness of each step. The implementation of the decomposition for different PDP models will be discussed in Section IV-F.

1) Why We Need Decomposition?: The reason for decomposition of the MAT is rather simple. Currently, the predication logic, compiled like "hard-coded logic" between MATs, is invisible to operators. Thus, even if the predication is wrongly programmed, there is no convenient way to identify the bug except manually reasoning the packet behavior.

As for match and action, they are closely coupled and concealed in one MAT. Consequently, operators can only review the results for the MAT instead of the behavior itself. In the real-world debugging case, knowing (*i*) which fields are matched for this packet and (*ii*) which actions and parameters are executed for this packet is significant for the operators to troubleshoot the bugs in the MAT. Thus, P4DB decomposes the MAT into three steps, and enables operators to see directly which behaviors are applied to which packet in the MAT.

2) Simulation of Single-step Debugging: Actually, although one MAT can be decomposed into three steps, P4DB cannot suspend the execution of the packet, and conduct the decomposed steps one by one. Thus in P4DB, we design a simulative way. Our design, although a simulative one, is reasonable. The basic purpose of P4DB is to let operators see what is wrong at runtime. In operating systems, the programmer can merely see the occurrence of the bug only after the bug is being triggered.

However, in the network system, the recurrence of the bug in a P4 program can be triggered by millions of packets in the same flow. In other words, there is no need to follow the execution of one packet in order to see what is happening. Instead, P4DB focuses on the P4 program itself and views packets as triggers for bugs. As long as the bug can be triggered by the consistent packets in the same flow, P4DB can provide operators with a simulative single-step debugging. And we will talk about this feasibility of bug recurrence in Section VII.

C. Debugging Primitives

As shown in Figure 3, we design a suite of user-friendly debugging primitives, so that operators can use these primitives to debug the P4 program interactively via the CLI. Debugging primitives are implemented by various compositions of debugging snippets. For now, there are seven fundamental primitives; however, with flourishing of debugging snippets, there will be more debugging primitives to cut down the debugging costs.

The attach and detach primitives can dynamically attach/detach the debugger to the instance of the P4 program on some specific device. One program can only be debugged by one CLI context, although operators can open multiple CLI contexts to debug different programs at the same time. When operators issue the attach primitive, P4DB will load the source code of the P4 program; analyze the source code; check the specific PDP model on the hardware device and prepare for the debugging environment.

The watch primitive provides operators with two different levels of visibility. One is the network-level visibility, which will show the network path of the specified flow. Another is the device-level visibility, which will show the MAT path of the specified flow. Internally, P4DB inserts a number of 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 the operator issues the break primitive, P4DB internally 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 into the decomposed MAT. Then operators can observe the states of predication. 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. Other primitives such as rmbp and show are trivial to be thoroughly described.

D. Debugging Snippets

1) Design of Debugging Snippets: The debugging snippet, usually composed of one or more MATs, matches the flow specified by operators and reports states of programmable elements to the debugging platform. The match rules in debugging snippet are instantiated by P4DB based on the parameters in debugging primitives. The actions in debugging snippets are

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 *valule* 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_meta_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

Figure 3. Debugging primitives.

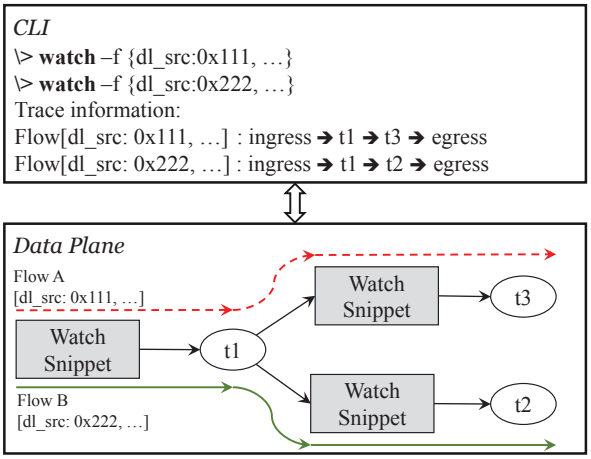


Figure 4. Design of watch snippets.

set to report different programmable elements according to the types of debugging snippets. Once the debugging snippet is deployed on the data plane, it will match the flow, and use the *generate_digest* action to send the reporting traffic to the debugging platform. For now, there are five types of debugging snippets.

Watch Snippet: The watch snippet is implemented by one MAT. The match rules in the watch snippet are set 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 shown in Figure 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.

Break Snippet: The break snippet, together with the predication snippet, match snippet and action snippet enables operators to debug the MAT in a fine-grained way as shown in Figure 5. When an operator issues the break primitive for one MAT, P4DB will decompose the MAT and install the break snippet. The break snippet is implemented by one MAT and reports the data plane states, including packet header, `metadata`, etc., to the debugging platform when triggered by the specified flow.

Predication Snippet: The predication snippet, between the predication step and the match step, is implemented by one MAT to report programmable elements that are referenced in the predication expression. If the original MAT does not have any predication expression, the predication step will do nothing but pass the flow to the match step. Notably, the specified flow will firstly be matched in predication step, then be passed to the predication snippet, while the normal flow will not be passed to the predication snippet. In the CLI, P4DB presents the predication expression as well as values to referenced variables. Hence, operators can check whether

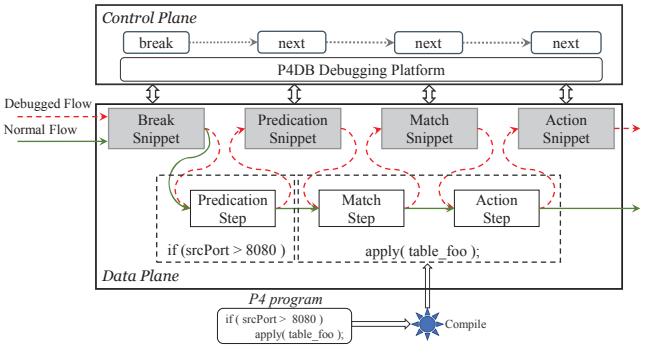


Figure 5. Design of break-predication-match-action snippets.

the boolean expression in predication is correct.

Match Snippet: The match snippet, implemented by one MAT, reports the `match fields` and values for the specified flow in the match step. By the match snippet, operators can inspect which `table entry` is matched by the specified flow, and verify correctness of match rules. The match snippet also merely processes the debugged flow.

Action Snippet: The action snippet, implemented by one MAT, reports packet headers, actions and action parameters to the debugging platform. By the action snippet, operators can verify whether the specified flow is correctly executed 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 action snippet to report which actions and parameters have been taken in the action step. The match rules and actions in the action snippet are instantiated when P4DB installs the action snippet.

2) Management of Debugging Snippets: P4DB adopts two ways of implementation for managing (installation/deletion) the debugging snippets. One is the on-demand way, another is the proactive way. Both implementations 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 the second next primitive. In P4DB, the break snippet, predication snippet, match snippet and action snippet are designed in this way. This way can guarantee that debugging snippets are always triggered by the latest inbound traffic and present on-data-plane states in real time. Besides, this way also causes a small performance overhead, since it does not need to install all relevant debugging snippets into the P4 program.

As for the proactive way, P4DB will install all related debugging snippets once the operators issue the debugging primitive. The watch snippet is designed in this way. When operators issue the watch primitive, P4DB will install the watch snippet for every MAT in the P4 program. This way may suffer from a performance overhead of multiple debugging snippets running on the data plane. However, it can collect much more data plane states in case the specified flow is too short to be consistently debugged.

E. Performance Optimization

In a network, millions of packets may pass through the data plane in every second. Since P4DB provides operators with live debugging ability by using on-data-plane debugging snippets, how to reduce the performance overhead becomes an important task. Actually, the purpose of the debugging snippets is to (*i*) filter the specified flow and (*ii*) send the reporting traffic. Accordingly, we propose two designs to reduce the performance overhead in terms of filtering and reporting respectively.

1) Placement of Filtering: The placement of filtering will directly impact the performance of the data plane. In P4DB, two ways of placement are adopted in consideration of trade-offs for different debugging snippets.

One is to place the filtering 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 chosen flows will be passed to the debugging snippet, and other normal traffic will bypass the debugging snippet. From Figure 5, we can see that the predication snippet, match snippet and action snippet adopt this way. This way of placement can effectively reduce the performance overhead, although it has a limitation in expressing the filtering rules. Since it requires that the debugged flow can be exactly matched by the MAT outside the debugging snippet.

Another is to place the filtering rules inside the debugging snippet, and use the debugging snippet itself to filter the specified flow. In this way, all traffic, no matter whether it is being debugged or not, will pass through both the debugging snippet and the original procedure. As shown in the Figure 4 and Figure 5, the watch snippet and break snippet adopt this way of implementation. Although this way of implementation may impose an extra overhead for filtering, it offers more flexibility in customizing matching rules for various debugged flows.

2) Message Damper for the Reporting Traffic: The reporting traffic also directly impacts the performance of the data plane and control channel. A large volume of reporting traffic necessarily incurs high CPU usage in switch as well as the congestion in the control channel.

Therefore, we design an on-data-plane message damper to suppress the reporting traffic. The message damper is implemented by periodically sampling packets in the debugged flow. The message damper maintains an adjustable threshold and a loop counter for each debugged flow. The threshold defines the period for sampling, while the loop counter counts the number of 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 greatly reduces the performance overhead in the data plane, meanwhile maintains the debugging functionalities of P4DB.

F. Implementation on Different PDPs

P4DB hides the heterogeneity of underlying PDPs and provides operators with a unified interface of debugging. This generality is internally implemented by maintaining one PDP driver for every type of PDP. In this paper, we respectively implement P4DB on a P4-specific PDP and a hypervisor-specific PDP. 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 PDPs.

In the P4-specific PDP, the predication and the MAT are closely coupled to the `control flow`. Therefore, in order to implement the decomposition of the MAT, we use two techniques. (*i*) As for the predication step, we manage to use a pipeline of two delicately designed MATs to represent the function of the predication expression equivalently. Based on this technique, the predication expression can be abstracted and equally expressed by the predication step. (*ii*) As for the match step and action step, the We add a redundant 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, since the execution of one MAT is already decomposed based on their designs, therefore, P4DB can be readily implemented on the hypervisor-specific PDP.

V. Debugging Workflow of P4DB

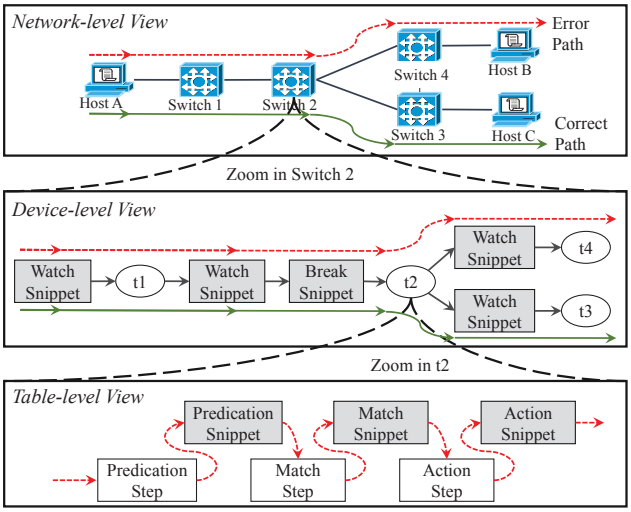


Figure 6. The debugging workflow of P4DB

In this section, we will illustrate the debugging workflow of P4DB through a real-world example shown in Figure 6. The debugging platform of P4DB maintains up-to-date status of all running P4 instances and devices. *Host A* is sending traffic to *Host C*. However, the traffic always goes through the error path (shown as the red dashed line) rather than the correct path (shown as the green line). The operator has checked the control plane applications and table entries in the data plane, and has found nothing wrong. Therefore, the operator starts to debug the running instance on *Switch 2* to find out the bugs. In the following part, we will describe the debugging workflow step by step.

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

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

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

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

#5: Afterwards, the operator issues the next primitive to check the predication logic. CLI shows the original predication expression which is none for $t2$. Nothing is wrong in this step.

#6: Then the operator again issues the next primitive 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 $t2$.

#8: The operator checks the 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, operator uses the detach primitive to stop debugging. The P4DB internally removes all debugging snippets on the data plane.

Actually, the bugs, like calling dependent `primitive actions` with a wrong order, are not rare in current P4 programming. Since, the execution pattern of `primitive actions` is defined as the serial pattern and the parallel pattern respectively in two versions of P4 language specification (`P4` and `P4-16`). Thus, without the full knowledge of hardware implementation and specification, operators are more prone to making these mistakes.

VI. Evaluation

A. Overview

We implemented P4DB on ONOS controller [23], and evaluated the performance of P4DB on the P4-specific PDP and the hypervisor-specific PDP accordingly. As previously mentioned, there are two ready-made hypervisor-specific PDPs: MPVisor and Hyper4. We choose MPVisor to conduct our experiments, for its improvement on performance and resource efficiency. The source code of P4DB can be found at https://P4DB.github.io/.

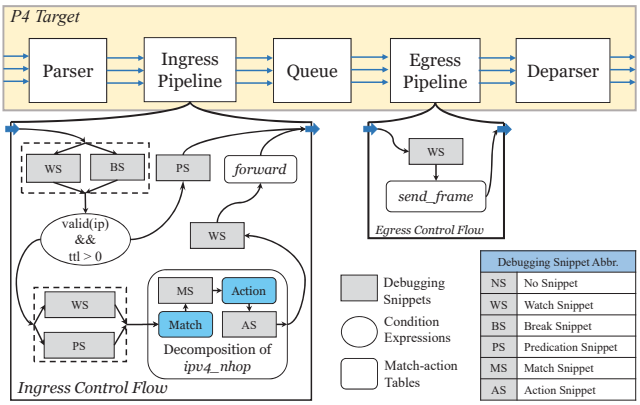


Figure 7. The router program with debugging snippets.

Our experiments are conducted on two x86 servers, each of which has 2×4 Intel E5-2637 CPU 3.50Ghz cores and 64GB memory. At the time of writing this paper, we do not have a P4 hardware device, thus, we use BMv2 as our device target for MPVsior and P4-specific PDP. The ONOS and BMv2 run on different servers. In our evaluation, we use the classic $router.p4$ program as the P4 program under test. As shown in Figure 7, $router.p4$ has three MATs and one predication expression. And we will install five kinds of debugging snippets into the program.

Since P4DB will insert debugging snippets into the original router program and lengthen the pipeline, it inevitably influences the throughput and delay of the router. Besides, the debugging snippets can generate a large volume of reporting traffic, and may congest the control channel and exhaust the CPU resource in the control plane. Therefore, we evaluate the performance of P4DB via three aspects: (*i*) data plane, (*ii*) control plane and (*iii*) control channel.

B. Performance of the Data Plane

As for the data plane, we take three groups of tests to demonstrate how different determinants can impact the performance. All three groups of tests are measured based on MPVisor and the P4-specific PDP respectively. Within each group, we conduct two benchmark tests respectively in terms of throughput and round trip delay. In each benchmark test, we add the performance of $router.p4$ running without P4DB as the baseline to see the performance overhead. The metrics for each group are as follows:

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

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

Group 3 Performance benchmarks in terms of different damper thresholds with message damper enabled

1) Analysis of Group 1: Table I shows the performance benchmarks with different numbers of debugging snippets on two PDPs.

Throughput: Both P4-specific PDP and MPVisor incur a throughput degradation as the number of debugging snippets increases. However, the degradation on MPVisor is much faster than P4-specific PDP. Since the internal control logic and model decoupling in MPVisor is rather complex, it amplifies the impact of the debugging snippets.

Table I
THE PERFORMANCE BENCHMARKS WITH DIFFERENT NUMBERS OF WATCH SNIPPETS ON P4-SPECIFIC PDP AND HYPERVISOR-SPECIFIC PDP

No. of watch snippets	P4		MPVisor	
	Throughput	Delay	Throughput	Delay
0 (baseline)	918.3 Mbps	0.278 ms	333.8 Mbps	0.331 ms
1	917.7 Mbps	0.300 ms	217.5 Mbps	0.352 ms
2	901.9 Mbps	0.291 ms	201.7 Mbps	0.388 ms
3	867.2 Mbps	0.309 ms	180.8 Mbps	0.406 ms
4	791.6 Mbps	0.311 ms	163.7 Mbps	0.418 ms

Delay: The number of debugging snippets seems to have little influence on the P4-specific PDP. The fluctuation of the results mainly comes from the indeterministic factors in BMv2's implementation. Unlike the processing delay on P4-specific PDP, the number of watch snippets do have an impact on the processing delay of MPVisor. As previously stated, MPVisor provides the uninterrupted feature, while suffers from the overhead for model translation.

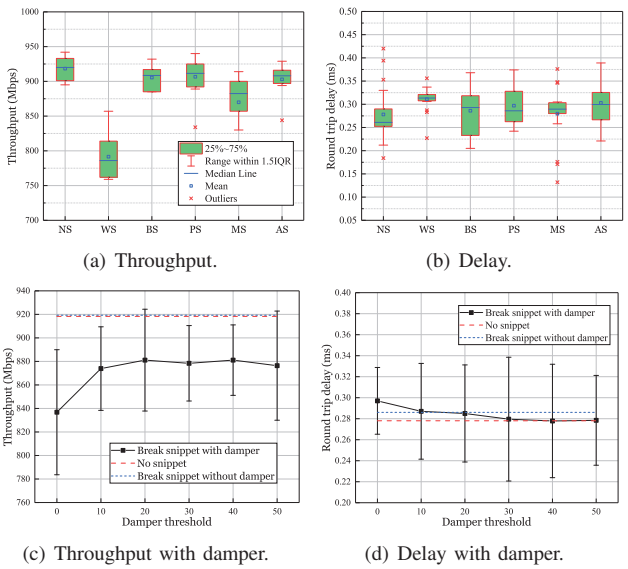
2) Analysis of Group 2: Figure 8(a), Figure 9(a); Figure 8(b) and Figure 9(b) show the performance benchmarks with different types of debugging snippets on two PDPs. In these figures, the abbreviations such as NS, MS, etc., are illustrated in the abbreviation table shown in Figure 7. Notably, NS denotes the baseline performance. To control variables in the experiments, we only install one type of snippets at one time.

Throughput: Figure 8(a) and 9(a) show throughput with different types of debugging snippets. The router program with four watch snippets performs worst. 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 MPVisor. However, on the P4-specific PDP, the other types of debugging snippets only have a performance degradation of few percents. Besides, the throughput penalty on MPVisor is larger than on P4-specific PDP, and ranges from 32.7% to 38.5% comparing with its baseline.

Delay: As shown in Figure 8(b) and 9(b), results are still quite different on respective PDPs. As for the P4-specific PDP, there is little difference among the baseline and various debugging snippets in terms of delay. However, for MPVisor, the increase of the delay ranges from 15% to 26% comparing with the baseline.

3) Analysis of Group 3: In this group, we test the router program with one break snippet as well as with the message damper enabled, to further inspect how the message damper impacts the performance. In the experiment, throughput and delay will be measured with different damper thresholds. Notably in these figures, the router without P4DB which is denoted as "no snippet" and the router with one break snippet but without damper which is denoted as " the break snippet without damper", are respectively depicted as two baselines for a comprehensive comparison.

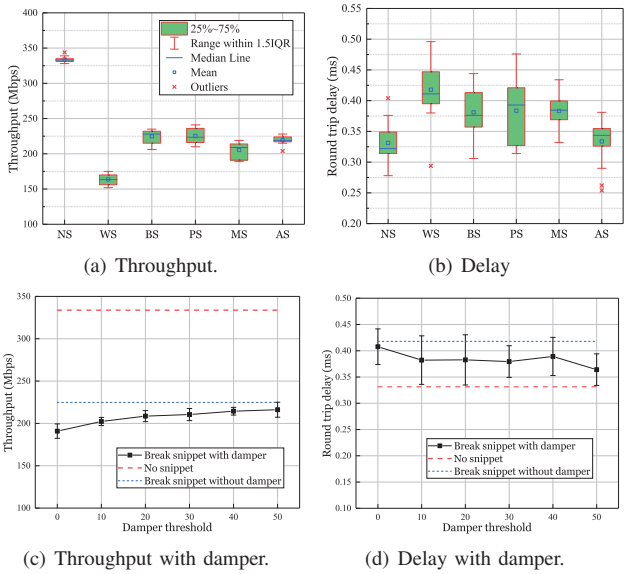
Throughput with the damper enabled: As shown in Figure 8(c) and 9(c). The throughput increases as the threshold



(a) Throughput. (b) Delay.

(c) Throughput with damper. (d) Delay with damper.

Figure 8. Performance and overhead of debugging snippets running on the P4-specific PDP.



(a) Throughput. (b) Delay

(c) Throughput with damper. (d) Delay with damper.

Figure 9. Performance and overhead of debugging snippets running on the hypervisor-specific PDP.

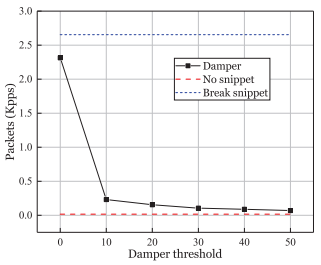


Figure 10. Debugging messages in the control channel.

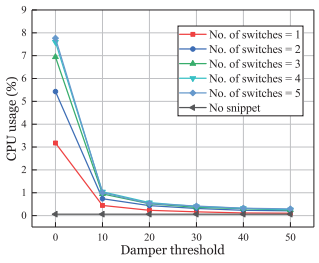


Figure 11. CPU usage with different numbers of debugged switches.

becomes larger. However, even if the threshold reaches as many as 50, the throughput is still smaller than the throughput of "the break snippet without the damper". Besides, on the P4-specific PDP, when the threshold is larger than 10, the increasing of throughput tends to be more stable. While, on MPVisor, the increasing of throughput becomes stable after the threshold is larger than 30.

Delay with the damper enabled: As shown in Figure 8(d) and 9(d). Overall, the message damper has a little positive effect on the processing delay. As the damper threshold increases, the delay deceases. On P4-specific PDP, the delay can even be the same as the baseline delay of "no snippet". However, the effect of the threshold is mitigated on MPVisor.

4) Summary: The performance of P4DB is quite different on P4-specific PDP and MPVisor. The difference can be mainly explained by the performance degradation caused by MPVisor itself. As previously mentioned in Section III-B, although MPVisor provides some *special debugging features* such as the uninterrupted reconfiguration of the PDP, it suffers from the performance overhead of model translation. On the other hand, on P4-specific PDP, P4DB provides a minor performance overhead and faces interruption of the data plane, even though the interruption time is usually under 50 ms according to [20].

Debugging snippets are inserted into the pipeline to improve the visibility of the data plane, and inevitably lead to a performance overhead. Since for runtime debugging, there is a trade-off between runtime visibility and performance. However, we think introducing the promising ability of on-the-fly debugging of runtime bugs with a certain degree of performance degradation is worthy especially in the case of network errors.

C. Control Channel and Control Plane

As for control channel and control plane, since they are agnostic to the underlying PDPs, we accordingly evaluate the (*i*) debugging message in the control channel and (*ii*) the CPU usage in the control plane.

Firstly, we measure the number of packets in the control channel in terms of different damper thresholds. The rate of the background traffic is 2Kpps. Secondly, we profile the runtime CPU usage to show that P4DB does not impose the substantial burden onto the controller with the help of the message damper. We measure the CPU usage in terms of different damper thresholds. Based on that, CPU usages with different numbers of debugged switches are also measured.

1) Analysis of Debugging Messages: As shown in Figure 10, the message damper has a sound effect on the number of packets in the control channel. When the damper threshold reaches 50, the number of packets generated by the break snippet is almost as same as the router with "no snippet". With the help of the message damper, P4DB can adaptively adjust the threshold and prevent debugging snippets from overwhelming the control channel.

2) 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 Figure 11, when the damper threshold is set to zero, the P4DB requires more CPU resource to process the workloads as the number of debugged 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. And the CPU usage declines to below 0.3% in all tested scenarios, when the damper threshold is set to 50.

3) Summary: P4DB imposes a certain overhead on the control channel and the control plane. However, operators can adaptively adjust the damper threshold to reduce the overhead from the reporting traffic generated by debugging snippets. And our experiments show that the damper performs well in terms of mitigating the pressure of control channel as well as the control plane.

VII. Discussion of Feasibility

As previously mentioned, P4DB implements the management of break, predication, match and action debugging snippets in an on-demand way, and using the latest inbound traffic as the trigger. Thus, it requires the flow to consistently trigger the runtime bugs while the operator is debugging the PDP. However, we consider this prerequisite of recurrence of runtime bugs is rather common. For example, according to [24], 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 really too short to be debugged, P4DB can readily collaborate with some log-based debugging tools to troubleshoot the bugs.

Another concern is about which kinds of bugs P4DB can deal with. Actually, P4DB intends to provide a general debugging platform based on different PDPs. P4DB focuses more on providing sets of convenient debugging primitives and entire visibility of data plane status for operators, thus is agnostic to the specific types of bugs.

VIII. Conclusion

This paper, for the first time, is devoted to the on-the-fly debugging of runtime bugs in P4-enabled networks. P4DB is proposed as a general debugging platform that empowers operators to debug diverse runtime bugs with simplicity and visibility. In P4DB, we propose three novel designs including (*i*) the debugging primitives, (*ii*) the debugging snippets, and (*iii*) the three-step decomposition of the MAT. Evaluation results show that P4DB enables operators to troubleshoot the runtime bugs in P4-enabled networks and only incurs a small performance overhead. In the future work, we plan to enrich the set of debugging primitives and develop automatic debugging applications based on the P4DB debugging platform.

IX. Acknowledgement

This research is supported by National Key R&D Program of China (2017YFB0801701) and the National Science Foundation of China (No.61472213). Jun Bi is the corresponding author. We gratefully thank all anonymous reviewers.

References

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

[2] Mojgan Ghasemi and et al. Dapper: Data plane performance diagnosis of tcp. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 61–74, New York, NY, USA, 2017. ACM.

[3] Vibhaalakshmi Sivaraman and et al. Heavy-hitter detection entirely in the data plane. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 164–176, New York, NY, USA, 2017. ACM.

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

[5] Nick McKeown and et al. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.

[6] Nikhil Handigol and et al. Where is the debugger for my software-defined network? In *Proceedings of the First Workshop on Hot Topics in Software Defined Networks*, HotSDN '12, pages 55–60, New York, NY, USA, 2012. ACM.

[7] Nikhil Handigol and et al. I know what your packet did last hop: Using packet histories to troubleshoot networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 71–85, Berkeley, CA, USA, 2014. USENIX.

[8] Ramakrishnan Durairajan and et al. Controller-agnostic sdn debugging. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 227–234, New York, NY, USA, 2014. ACM.

[9] Peng Zhang and et al. Mind the gap: Monitoring the control-data plane consistency in software defined networks. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 19–33, NY, USA, 2016. ACM.

[10] Q. Zhi and et al. Med: The monitor-emulator-debugger for software-defined networks. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[11] Andreas Wundsam and et al. Ofrewind: Enabling record and replay troubleshooting for networks. In *Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'11, pages 29–29, Berkeley, CA, USA, 2011. USENIX Association.

[12] Hongyi Zeng and et al. Automatic test packet generation. *IEEE/ACM Trans. Netw.*, 22(2):554–566, April 2014.

[13] Haohui Mai and et al. Debugging the data plane with anteater. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 290–301, New York, NY, USA, 2011. ACM.

[14] Peyman Kazemian and et al. Header space analysis: Static checking for networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[15] K. Bu and et al. Is every flow on the right track?: Inspect sdn forwarding with rulescope. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, April 2016.

[16] Peter Perešíni and et al. Monocle: Dynamic, fine-grained data plane monitoring. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '15, pages 32:1–32:13, New York, NY, USA, 2015. ACM.

[17] Nick McKeown and et al. Automatically verifying reachability and well-formedness in p4 networks. Technical report, September 2016.

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

[19] Sharad Chole and et al. drmt: Disaggregated programmable switching. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, pages 1–14, New York, NY, USA, 2017. ACM.

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

[21] David Hancock and et al. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '16, pages 35–49, New York, NY, USA, 2016. ACM.

[22] Cheng Zhang and et al. Mpvisor: A modular programmable data plane hypervisor. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 179–180, New York, NY, USA, 2017. ACM.

[23] Pankaj Berde and et al. Onos: Towards an open, distributed sdn os. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 1–6, New York, NY, USA, 2014. ACM.

[24] Hongyi Zeng and et al. A survey on network troubleshooting. Website. http://yuba.stanford.edu/~peyman/docs/atpg-survey.pdf.