Toward Consistent State Management of Adaptive Programmable Networks Based on P4

Mu He, Andreas Blenk, Wolfgang Kellerer
Chair of Communication Networks
Technical University of Munich

Stefan Schmid
Faculty of Computer Science
University of Vienna

ABSTRACT

Emerging network applications (augmented reality, industrial Internet, etc.) introduce stringent new requirements on the performance, dependability, and adaptability of communication networks. Programmable data planes (e.g., based on P4) provide new opportunities to meet these requirements, by enabling adaptive network reconfigurations. However, ensuring *consistency* during such reconfigurations remains challenging. This paper makes a first step toward a more automated state management of adaptive data planes. In particular, we present an efficient P4 state management framework, `P4State`, which allows to quickly identify the network states from the source code that are critical for data plane reconfigurations (e.g., due to scaling, failure recovery). We report on first promising evaluation results of our prototype implementation in terms of correctness and efficiency, also considering two case studies using HULA (load balancing in data center) and HashPipe (line-rate measurement in data plane).

CCS CONCEPTS

• **Networks** → *Programmable networks*; In-network processing; • **Software and its engineering** → *Operational analysis*;

KEYWORDS

Programmable Data-Planes, Stateful Reconfiguration, Code Analysis, P4

ACM Reference Format:
Mu He, Andreas Blenk, Wolfgang Kellerer and Stefan Schmid. 2019. Toward Consistent State Management of Adaptive Programmable Networks Based on P4. In *Proceedings of ACM SIGCOMM 2019 Workshop on Networking for Emerging Applications and Technologies (NEAT'19).* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3341558.3342202

1 INTRODUCTION

Emerging applications in the context of, e.g., smart city, autonomous driving, virtual reality, or robotics, to just name a few, revolutionize the way that humans interact with the physical world [28]. Such applications often involve large amounts of data and require tactile control loops, introducing stringent requirements on the underlying communication network: not only in terms of performance (e.g., throughput and latency) and dependability (e.g., fast failover), but also in terms of *adaptability*. The ability to *reconfigure* and *adapt* the network to the current workload (e.g., by scaling out/in resources in a demand-aware manner) or context (e.g., in terms of fast reactions to failures), allows to operate networks at new levels of efficiency.

Over the last years, the advent of the Programmable Data Plane (PDP) with P4 [7] assists flexible and dynamic slicing of end-to-end network and service resources [13, 28]. However, leveraging PDP toward efficient and *consistent* network reconfigurations remains challenging. In particular, many PDP operations are *stateful*: the forwarding devices maintain states for the network services, e.g., registers for forwarding port mapping and meters for traffic rate control. In order to provide a predictable data plane behavior, we need to ensure state *consistency*, at any time. For example, in an autonomous driving context, network services provisioned by the vehicles should be migrated together with their states (e.g., service ports and IP addresses), to guarantee connectivity and QoS.

This paper is motivated by the observation that a prerequisite to preserve state consistency is to automatically *recognize* all the states in a PDP program (e.g., P4) that should be maintained during reconfiguration: an aspect which to the best of our best knowledge has not been addressed in the P4 literature yet. State recognition however is challenging because of the wide range of possible notions of states (and access methods) in P4-based PDPs.

We present `P4State`, a first step toward automated P4 state management. Our framework includes an analyzer that takes a P4 program as input, collects state access along the packet processing pipelines, and visualizes the analysis output for P4 programmers and operators. To recognize the states, we apply *control-flow analysis* [21] on top of the Control Flow Graph (CFG), which is built out of the program. We prune the original CFG graph and only keep the nodes with state access, i.e., read or write, to make it more accessible. Finally, all stateful paths within the CFG with involved state entities are available as output and therefore can help to maintain consistency during network reconfiguration.

Contributions. Succinctly, our contributions are:

- We characterize the states of P4 data planes and provide a taxonomy of the usage of registers (which store states) in open-source P4 programs.
- We propose a suite of algorithms to analyze P4 programs and to identify the states that need to be maintained during reconfiguration.
- We report on the implementation and evaluation of the program analyzer, considering synthetic and real programs

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
NEAT'19, August 19, 2019, Beijing, China
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6876-6/19/08…$15.00
https://doi.org/10.1145/3341558.3342202

(namely HULA and HashPipe), and in terms of correctness and efficiency.

Organization. The remainder of this paper is organized as follows. Section 2 characterizes states in P4 data plane. Section 3 describes the algorithms to analyze the register accesses within a P4 program. Section 4 introduces the prototype of P4State and two case studies. We discuss related works in Section 5. In Section 6 we conclude the paper and discuss possible extensions of P4State.

2 CHARACTERIZING P4 PDP STATES

P4 is a domain specific language that can describe the packet processing behavior of programmable data plane [7]. The programmability exists in the parser, the match-action pipeline, and the deparser. The parser can extract any customized fields from the header, whereas the deparser insert values back to the fields in the end. The match-action pipeline consists of multiple stages, each with a matching table and customized action(s) of packet processing. P4 programs are supported in various data plane targets, such as BMv2 [3], SmartNIC [2] and NetFPGA [31].

The definition of states in a P4 data plane is quite broad [6, 22]. States include (i) table entries, (ii) stateful variables defined in the P4 specification, and (iii) (some) temporary variables defined in a program. We include the temporary variables only if they act like pointers that refer to stateful variables (details in Sec. 3.2). Since the table entries can be recognized and maintained by the control plane during data plane reconfiguration without much effort, we do not consider them in this paper.

The P4 specification defines three types of stateful variables: *register*, *meter*, *counter*. The variables need to be persistent, i.e., their values should persist beyond a single iteration of the packet processing loop [21]. In this paper, we focus on the *register* variable, as it is commonly adopted in real P4 programs. Regarding meter and counter, we briefly discuss the mechanism to maintain their consistency in Sec. 6.

Register Usage. We identify the following scenarios when *registers* can be declared:

- a value that the processing of the following packets would access, e.g., packet counter[1] [29].
- a value that the control plane can access for making control decisions, e.g., the status of a port [27].
- a value that controls the packet processing in a P4 node, e.g., a flag enabling on-demand functionality [14].

The usage of a register is one of the factors indicating whether it should be transfered during data plane reconfigurations. As an example, Figure 1 shows the declaration (line 2) and read-access (line 5) of the content of register *flag_reg*. Note that in this example, we denote *flag_reg* as a **register type** and the content as a **register entry**. Table 1 gives an overview of the P4 programs that leverage registers in their implementation[2].

Register access within a P4 program can be either read or write (both directly and indirectly). In Figure 1, the binary register value decides the following packet processing path: either line 6 or line 7. This is an indirect access in an if-conditional, i.e., a temporary variable that refers to the value of a register entry is evaluated. Register entries can *only* be directly accessed in actions. Figure 2 demonstrates an example, where a register entry is read and copied to one field in the user-defined custom metadata. Actions are always associated with tables; an action is called either based on the matching result of a table or when a packet processing path traverses a table. Meanwhile, the value of a register entry can impact the decision of an if-conditional.

Register Classification. We classify registers into two categories: *flow-based* and *device-based*. A *flow-based* register saves per-flow state, and typically instantiates a large number of entries (e.g., 65536), which can be migrated on the data plane and control plane. A *device-based* register saves the state that is device-specific. It has less entries, but may need to be migrated with the help of the controller. The classification helps to coordinate the maintenance of various registers at runtime, i.e., decide how to migrate them.

Luo et al. [22] advocate that it is not necessary to migrate the flow-based register that is computed from the events of arriving

Table 1: Overview of Register's Usage in P4 Programs

Program Name	LoC	Types of Reg.	# Reg. Ent.
heavy hitter	178	2	32
flowlet	203	2	16384
netpaxos [8]	210	6	256002
ndp [12]	223	2	4
hashpipe [29]	229	8	224
hula [20]	289	4	65632
dapper [11]	535	22	86
sketchlearn [15]	646	32	8192
linearroad [17]	789	11	6096
netcache [18]	1427	40	6784

Note: LoC calculation based on programs written with P4-16 for the BMv2 simple switch target. Programs written in P4-14 are translated to P4-16.

```
1 control ingress(...) {
2   register<bit<1>>(32w1) flag_reg; ...;
3   apply {
4     bit<16> flag;
5     reg.read(flag, 0);
6     if (flag == 1) {...;}
7     else {...;}
8   }
9 }
```

Figure 1: Register declaration and indirect register access.

```
1 action read_register() {
2   hash(reg_index, HashAlgorithm.crc16, (bit<32>)0,
3   {...}, (bit<32>)65536);
4   reg.read(meta.custom_metadata.val, reg_index);
5 }
```

Figure 2: Direct access in an action.

[1]Similar to the usage of a native counter, but supported by more P4 targets.
[2]For a more detailed description of all P4 programs we have surveyed, please refer to https://github.com/muhe1991/p4-programs-survey.

packets, e.g., the *flowlet_id*, which denotes a flowlet and is calculated from the header field tuple. However, we argue that in order to maintain the consistency requirement, we have to migrate those flow-based registers. For example, when the *flowlet_id* determines the egress port, the loss of its values might lead to the following packets of the same flowlet to be forwarded on a different path; this can induce jitter and delays.

Register Migration. States can be migrated either through the data plane [22], where register values are carried as specific header fields, or migrated through the controller [14], which performs direct register read/write. For the first approach, a *flow-based* register entry needs an exact flow to piggyback its value, which can lead to long migration latencies when flow patterns change quickly and the expected flows do not show up on time. For the second approach, since the state values need to be copied first from the source node to the controller and then copied to the destination node, we need to take into account the extra forwarding time from the nodes to the controller. Note that we assume the controller integrates the functionalities of both the control plane and the management plane.

No matter which approach is applied, we can assume that the total migration time increases with the number of register entries, no matter which migration approach we would apply. Here the migration time is defined as the overall time spent for migrating all necessary states before the new node can work correctly. For the flow-based registers, i.e., the ones with indices calculated with hash functions, we may potentially create many (more than 2^{16}) entries, depending on an initial guess of how many flows can show up in the network. However, not all entries are filled with effective values that need migration. In other words, an intuitive approach that strives to migrate all register values would induce a long total migration time. Therefore, we propose `P4State` which recognizes only necessary register values.

3 P4STATE: DESIGN AND ALGORITHMS

In this section, we describe the algorithm set to analyze the register accesses of a P4 program. Instead of directly parsing the P4 code, we leverage the P4 compiler to produce a compiled `.json` file and feed it to the algorithm suite [23]. The basic idea is to identify state accesses in tables and conditionals (Sec. 3.1 and 3.2), translate the P4 program into a Control Flow Graph (CFG) (Sec. 3.3), delete all nodes in the CFG without references to registers (Sec. 3.4), traverse all paths in the CFG to collect register accesses, and conclude the registers that need to be migrated (Sec. 3.5). The CFG represents all paths that might be traversed in a program during its execution.

3.1 Identifying States

As first step, we collect all the declared registers (including their depths and widths) and classify them as *flow-based* or *device-based*. The classification criterion is the width of the register. As a common practice, a flow is identified by a hash value with width 16 or 32 [1], which corresponds to hashing algorithms defined in v1 model [5] (CRC16 or CRC32). Therefore, a register whose width is larger or equal than 16 is classified as *flow-based* register. In order to be comprehensive, we also classify the registers, whose indexes are calculated with hash functions, as *flow-based* registers. Those that are not classified as *flow-based* registers will be treated as *device-based* registers.

Algorithm 1: Register Binding

```
   Data: p4prog.json
   Result: set of registers R, headers H, actions A, tables T and
           conditionals C
 1 Fill R, H and A;
 2 T = ∅, C = ∅; // Initiate set of tables and conditionals
 3 Extract ingress/egress pipeline;
 4 for t in pipeline do
 5 |   R_t = ∅;                              // for all tables
 6 |   for a in A_t do
 7 |   |   for r in a_r do
 8 |   |   |   R_t = R_t ∪ {r};
 9 |   T = T ∪ {t};
10 for c in pipeline do
11 |   R_c = ∅;                          // for all conditionals
12 |   for h in associated headers do
13 |   |   Update set of registers R_c through h_r;
14 |   C = C ∪ {c};
```

3.2 Table/Conditional Register-Binding

Algorithm 1 traverses all tables and if-conditionals defined in the `.json` file and associates the registers that are accessed by them. In order to do this, it first collects all defined registers, headers, actions, tables and conditionals in the pipelines.

Since direct register access inside conditionals is not possible, it is non-trivial to associate registers to conditionals. We first collect all header fields (including the temporary variables), e.g., *flag* in Figure 1 and *reg_index* in Figure 2. Afterwards, we traverse all statements in actions and associate the header field with the register when there is a register access. Note that the statements in the *apply* struct are translated into an action associated with a table. The temporary variable *flag* is declared as a custom header field *scaler_flag*.

Besides the set of all registers R, we also fill the set of accessed registers for each table t and conditional c. For a table t, we check all associated actions and place every accessed register in the set R_t (see line 5-8). Similarly, for a conditional c, we check all associated headers and place every accessed register in the set R_c.

3.3 Stateful CFG Construction

As mentioned before, the packet processing pipeline (i.e., the CFG) described by P4 can be decomposed as a bunch of basic entities (nodes) of tables and conditionals. A table has only one egress, whereas a conditional has two, each associated with a decision result (True/False).

We construct a CFG of the P4 program under analysis with Algorithm 2. Following the pattern of p4c-graphs [4], the processing path of each packet always starts from "START", traverses different sets of tables and conditionals, and terminates at "EXIT". The introduction of START and EXIT (as dummy tables) provides the two anchor points for all possible processing paths. After adding the first edge between START and the initial node (table or conditional),

Algorithm 2: Graph Formulation (GraphForm)

Data: Set of R, H, A, T, C
Result: CFG
1 Initiate CFG as an empty directed graph;
2 Add node START and EXIT as two dummy tables;
3 Extract the first node n_1 from the pipeline;
4 Call AddEdge(CFG, START, n_1);
5 $N = \{START\}$;
6 Call ExpNext(n_1, N);

Algorithm 3: Explore Next Node (ExpNext)

Data: current node n, node list N
1 $N = N \cup n$;
2 **if** $n \in T$ **then**
3   **if** *next node of* n *is EXIT* **then**
4         $N = N \cup EXIT$;
5   Call DrawPath(N);
6   **else**
7   **for** *each node* $\hat{n}$ *after* n **do**
8   Call ExpNext($\hat{n}$, N);
9 **else**
10   **for** *each next node* $\hat{n}$ *after* n **do**
11   **if** $\hat{n}$ *is EXIT* **then**
12             $N = N \cup EXIT$;
13   Call DrawPath(N);
14   **else**
15   Call ExpNext($\hat{n}$, N);

it calls *ExpNext* to further explore the path until EXIT, which is described in Algorithm 3.

Algorithm 3 works in a recursive manner. It stops calling itself only if *(i)* there is no node after the current table (line 3-5), or *(ii)* there are no entities on the true or false branch of the current conditional (line 11-13). In this case, it calls *DrawPath* to draw a full path from START to EXIT.

3.4 CFG Pruning

The stateful CFG assists the following register accesses analysis. In our design, the analysis should be able to return all paths with state access. However, the original CFG and the paths inside can come with very large size, which is hard for humans to consume. Inspired by the idea of *Thin Slicing* [30], we exclude all stateless nodes from the CFG, in order to produce a human-friendly (pruned) version. The pruned CFG provides an evident view of all stateful operations.

The pruning process consists of two steps (described in Algorithm 4 and 5). The first step detects all nodes that do not have access to registers, i.e., $n_R == \emptyset$ in line 5, and removes these nodes. The nodes before any node to be removed, i.e., $\overline{n}$, and after it, i.e., $\underline{n}$, should be reconnected to ensure complete path(s) from START to EXIT. The second step merges consecutive tables on a single path, if they access the same registers, i.e., $\underline{n}_R == n_R$. Only the first table stays, whereas the following tables are replaced with edges in the graph.

Algorithm 4: Pruning - Stateless Node Elimination

Data: Original CFG
Result: Intermediate CFG
1 Call UpdateNeighbourNodes();
2 **for** *each node* n *in CFG* **do**
3   **if** n *is START or EXIT* **then**
4   continue;
5   **if** $n_R == \emptyset$ **then**   // n_R denotes the register set
6   Call RemoveNode(CFG, n);
7   **for** $\overline{n}$ *in* $\overline{N}_n$ **do**
8   **for** $\underline{n}$ *in* $\underline{N}_n$ **do**
9   Call AddEdge(CFG, $\overline{n}$, $\underline{n}$);
10   Call UpdateNeighbourNodes();
11 **for** *each conditional* c *in* C **do**
12   **if** c *in CFG* **then**
13   Get all paths P from START to c;
14   **for** $p \in P$ **do**
15             $\hat{R} = \emptyset$;
16   **for** *node* n *in* $p[1:-1]$ **do**
17                 $\hat{R} = \hat{R} \cup n_R$;
18   **if** $\hat{R} == \emptyset$ **then**
19   Call RemoveNode(CFG, c);
20   **for** $\overline{n}$ *of* c **do**
21   **for** $\underline{n}$ *of* c **do**
22   Call AddEdge($\overline{n}$, $\underline{n}$);

Algorithm 5: Pruning - Nodes Merging

Data: Intermediate CFG
Result: Pruned CFG
1 **for** *each node* n *in CFG* **do**
2   **if** n *is START or EXIT or not a table* **then**
3   continue;
4   **for** *next node* $\underline{n}$ *of* n **do**
5   **if** $\underline{n}$ *is not a table* **then**
6   continue;
7   **if** $\underline{n}_R == n_R$ **then**
8   Call RemoveNode(CFG, $\underline{n}$);
9   **for** *next node* $\underline{\underline{n}}$ *of* $\underline{n}$ **do**
10   Call AddEdge(CFG, n, $\underline{\underline{n}}$);
11   Call UpdateNeighbourNodes();

As function utilities, the method *UpdateNeighbourNodes* updates the previous and subsequent nodes of each node, given the current status of the CFG. The method *RemoveNode* removes one node and all edges connected to it.

3.5 Path & Role Identification

Finally, the analyzer recognizes all paths with state access and generates the pruned CFG as well as a report listing all stateful paths and their respective associated state sets. When a P4 program consists of multiple functions, which are enabled/disabled upon startup (e.g., HULA [1]), the analyzer can also infer such information and report the enabled functionalities. For this, it leverages both the pruned CFG and the controller rules such as register initializations (typically specified in a file). If the controller maintains the state

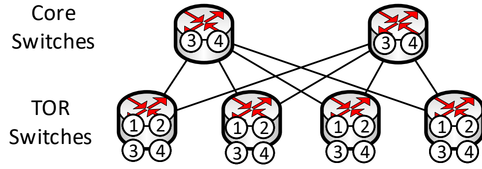


Figure 3: An exemplary topology of HULA with circled numbers representing different types of registers.

consistency, such information can also assist with deciding the order in which different types of states should be transferred.

4 PROTOTYPE AND EVALUATION

The prototype of `P4State` mainly includes a code analyzer and the utilities of P4C compiler [4]. We implement the analyzer in Python with 500 LoC [3]. `P4State` takes a P4 program as input, analyzes its compiled `.json` format, and outputs the paths with state accesses.

We provide a first impression of `P4State`'s practicability with case study on two real P4 programs. Afterwards, we evaluate the efficiency of our propose algorithms with both real and synthetic programs.

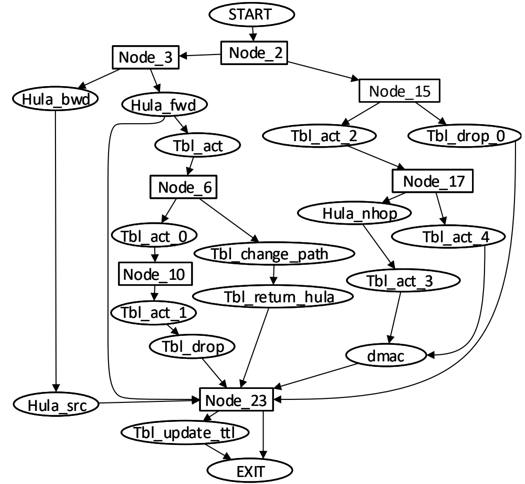
4.1 Case Study

HULA. HULA addresses congestion in data center networks. For this, HULA switches run two functions: probing and forwarding. Figure 3 depicts an exemplary data center topology with HULA. Probing is deployed on the ToR switches for finding best paths in the core. The probing updates the forwarding function, which then forwards all data plane traffic. `hula.p4` [1] is a simplified version of HULA with four types of registers: ① *srcindex_qdepth_reg*, ② *srcindex_digest_reg*, ③ *dstindex_nhop_reg* and ④ *flow_port*. ① and ② store the queue length and the digest of the best path from each ToR. ③ keeps the next hop to reach each ToR, and ④ keeps the next hop for each flow.

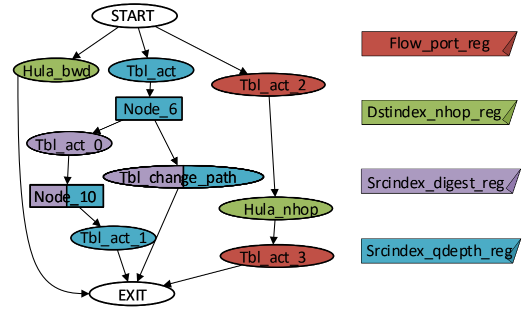
`hula.p4` merges the pipeline of probing and the pipeline of forwarding in one program. To decide which pipeline should be referred to for a single packet, the program checks the *hula* header field of the received packet. For the ToR switches, all four types of registers are needed to enable HULA update and normal packets switching. For the non-ToR (i.e., core) switches, only type ③ and ④ are needed. `P4State` successfully recognizes the above two functions, and outputs the pruned CFG with 12 nodes (original CFG 25 nodes), which is shown in Figure 4. Such knowledge can help to maintain state consistency during data plane reconfiguration. For example, when a core switch is about to fail, migrating states of type ③ and ④ to a backup switch would be sufficient to ensure that all current best path in the core are preserved.

HashPipe. To perform line-rate measurements in the data plane, HashPipe [29] implements a pipeline of hash tables to record heavy flows, i.e., flows with a huge number of packets. There are 8 types of registers that come within total 224 entries. The flows are tracked and the tracking information is maintained within three types of registers (two stages, in total six). One type is used to track the flow

[3]We plan to make the code public in an extended version of this paper.



(a) Original CFG



(b) Pruned CFG

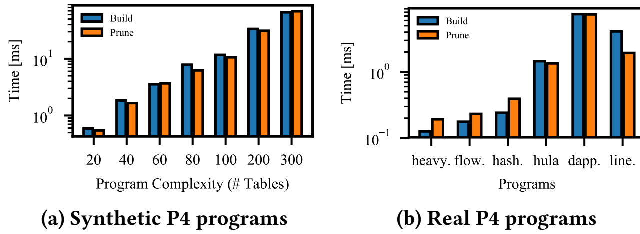
Figure 4: Control Flow Graph (CFG) of `hula.p4`: ellipses represent tables, squares represent conditionals, and arrows represent packet processing paths. Colors indicate the accesses of respective register types.

identifiers (source IPv4 addresses), and another type is used to store the packet counts corresponding to the identifiers. The last type shows whether each table entry is valid, i.e., there are non-zero values for the previous two types of registers. `P4State` recognizes only one function, i.e., counter update, and all registers that needed to be migrated correctly.

4.2 General Performance Measurement

To evaluate the efficiency of `P4State`, we measure the runtime of the CFG construction module and the CFG pruning model, which together account for the most algorithmic execution time. The measurement is performed on both synthetic and real P4 programs. We use Whippersnapper [9] to generate synthetic programs having 20 to 300 tables. The realistic programs are selected from Table 1.

Figure 5a presents the runtimes when analyzing synthetic programs; each data point is the average of 30 measurement runs. It shows comparable runtimes of CFG construction and pruning: they increase exponentially with the number of tables. Nevertheless, a



(a) Synthetic P4 programs **(b) Real P4 programs**

Figure 5: Runtime of CFG construction and pruning.

program with 300 tables (which is more than all realistic P4 programs that we have collected) can be analyzed in around 100 ms. Figure 5b presents the results for real programs, which we sort according to the program's complexity (represented as LoC) along the x-axis. We observe that the code analysis takes up to 15 ms for the CFG construction and pruning (the case of `dapper.p4`). Even though `linearroad.p4` has the highest LoC, its analysis time is not necessarily the highest, due to its simpler pipeline structure. In conclusion, the analyzer is very efficient.

5 RELATED WORK

To the best of our knowledge, we are the first to study the consistent state management in an elaborated manner for P4 data plane. However, there is much interesting previous work on state management in the context of general NFV and P4, as well as analyzing P4 code.

Data Plane State Management. The research of state management of data plane is quite abundant. Split/Merge [26] requires middleboxes to allocate and access all states through a customized shared library. OpenNF [10], however, transfers directly the serialized states between different middleboxes. From a different perspective, StatelessNF [19] requires middleboxes to create/read/update states in a central data store, which allows any middlebox to access any state at any time. SNAP [6] considers state allocation in a static scenario; the whole network is considered as a single switch and the location of states in the form of forwarding rules are optimized to enforce policy.

State Management of P4. SwingState [22] initiates the study of state transfer during reconfiguration by piggybacking states on the data plane packets. P4NFV [14] recommends managing the state with the controller, which has a holistic view of the data plane states and can perform operations such as merging on the states during reconfiguration.

NFV Program Analyzer. Many tools were proposed to analyze data plane program for performance or security. CASTAN [25] and BOLT [16] discover execution paths within the code of an NF and recognize potentially large resource consumptions, e.g., CPU cycles and memory accesses. P4pktgen [24] analyzes a P4 program and generates input packets and table entries that cover all execution paths. Assert-P4 [23] leverages assertions and symbolic execution to validate the general network correctness properties. However, the analysis for data plane reconfiguration is still missing in the literature.

6 CONCLUSION & DISCUSSION

Motivated by the need for more adaptive networks and the challenge of consistent reconfigurations of stateful data planes, we presented, implemented, and evaluated `P4State`, an automated mechanism to recognize states in P4. `P4State` is able to quickly analyze programs and successfully recognizes the register types that need migration during data plane reconfiguration. With synthetic and real programs, we show the efficiency of our proposed algorithms in terms of runtime.

We understand `P4State` as a very first step, but believe that it readily provides many interesting and promising extensions, which we discuss as follows.

Line-rate Processing and Verification. `P4State` outputs an overview of multiple register accesses in a P4 program. With that, we can identify when a read or a write operation is performed more than one time to the same register type, which can lead to longer processing times [29]. Moreover, the analyzer can automatically detect race conditions of register access and write-before-read error.

Group Transfer. Currently the controller only accesses one register entry at a time. If multiple entries can be transferred simultaneously, the forwarding latency can be greatly reduced. In that case, `P4State` can be extended to detect the valid entries that will be transferred all at the once.

Counters & Meters. Since the data plane cannot read counters, the control plane reads and stores all values before reconfiguration, and if possible, updates the counted values afterwards. For the meters, the control plane is always in charge of their configurations, therefore the controller only needs to configure the previous meter settings upon startup of new data plane entity. `P4State` can be extended to recognize counters and meters and facilitate the maintenance of them during runtime.

Consistent Network Update. Updating a network policy can involve the reconfiguration of multiple P4 nodes which is not trivial. It would be interesting to investigate the order of state update in a multi-node P4 environment, to avoid data plane misbehaviors (e.g., routing loops and black-holes) during policy update.

ACKNOWLEDGMENT

This work is part of a project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647158 - FlexNets). The authors alone are responsible for the content of the paper.

REFERENCES

[1] 2019. HULA. https://github.com/p4lang/tutorials/tree/sigcomm_17/SIGCOMM_2017/exercises/hula. (2019).
[2] 2019. Netronome SmartNIC. https://www.netronome.com/blog/p4-programmability-for-the-netronome-agilio-smartnic/. (2019).
[3] 2019. P4 behavioral-model. https://github.com/p4lang/behavioral-model. (2019).
[4] 2019. P4C. https://github.com/p4lang/p4c. (2019).
[5] 2019. V1 Model. https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4. (2019).
[6] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. 2016. SNAP: Stateful network-wide abstractions for packet processing. In *SIGCOMM*. ACM, 29–43.
[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *SIGCOMM CCR* 44, 3 (2014), 87–95.

[8] Huynh Tu Dang, Daniele Sciascia, Marco Canini, Fernando Pedone, and Robert Soulé. 2015. Netpaxos: Consensus at network speed. In *SOSR*. ACM, 5.

[9] Huynh Tu Dang, Han Wang, Theo Jepsen, Gordon Brebner, Changhoon Kim, Jennifer Rexford, Robert Soulé, and Hakim Weatherspoon. 2017. Whippersnapper: A p4 language benchmark suite. In *SOSR*. ACM, 95–101.

[10] Aaron Gember-Jacobson, Raajay Viswanathan, Chaithan Prakash, Robert Grandl, Junaid Khalid, Sourav Das, and Aditya Akella. 2014. OpenNF: Enabling innovation in network function control. In *SIGCOMM CCR*, Vol. 44. ACM, 163–174.

[11] Mojgan Ghasemi, Theophilus Benson, and Jennifer Rexford. 2017. Dapper: Data plane performance diagnosis of tcp. In *SOSR*. ACM, 61–74.

[12] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-architecting datacenter networks and stacks for low latency and high performance. In *SIGCOMM*. ACM, 29–42.

[13] Mu He, Alberto Martínez Alba, Arsany Basta, Andreas Blenk, and Wolfgang Kellerer. 2019. Flexibility in Softwarized Networks: Classifications and Research Challenges. *IEEE Comm. Surveys & Tutorials* (2019).

[14] Mu He, Arsany Basta, Andreas Blenk, Nemanja Deric, and Wolfgang Kellerer. 2018. P4NFV: An NFV Architecture with Flexible Data Plane Reconfiguration. In *CNSM*. IEEE, 90–98.

[15] Qun Huang, Patrick PC Lee, and Yungang Bao. 2018. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. In *SIGCOMM*. ACM, 576–590.

[16] Rishabh Iyer, Luis Pedrosa, Arseniy Zaostrovnykh, Solal Pirelli, Katerina Argyraki, and George Candea. 2019. Performance contracts for software network functions. In *NSDI*. 517–530.

[17] Theo Jepsen, Masoud Moshref, Antonio Carzaniga, Nate Foster, and Robert Soulé. 2018. Life in the fast lane: A line-rate linear road. In *SOSR*. ACM, 10.

[18] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. 2017. Netcache: Balancing key-value stores with fast in-network caching. In *SOSP*. ACM, 121–136.

[19] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. 2017. Stateless network functions: Breaking the tight coupling of state and processing. In *NSDI*. USENIX, 97–112.

[20] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. 2016. Hula: Scalable load balancing using programmable data planes. In *SOSR*. ACM, 10.

[21] Junaid Khalid, Aaron Gember-Jacobson, Roney Michael, Anubhavnidhi Abhashkumar, and Aditya Akella. 2016. Paving the Way for NFV: Simplifying Middlebox Modifications Using StateAlyzr. In *NSDI*. USENIX, 239–253.

[22] Shouxi Luo, Hongfang Yu, and Laurent Vanbever. 2017. Swing state: Consistent updates for stateful and programmable data planes. In *SOSR*. ACM, 115–121.

[23] Miguel Neves, Lucas Freire, Alberto Schaeffer-Filho, and Marinho Barcellos. 2018. Verification of P4 programs in feasible time using assertions. In *CoNEXT*. ACM, 73–85.

[24] Andres Nötzli, Jehandad Khan, Andy Fingerhut, Clark Barrett, and Peter Athanas. 2018. P4pktgen: Automated test case generation for p4 programs. In *SOSR*. ACM, 5.

[25] Luis Pedrosa, Rishabh Iyer, Arseniy Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. 2018. Automated synthesis of adversarial workloads for network functions. In *SIGCOMM*. ACM, 372–385.

[26] Shriram Rajagopalan, Dan Williams, Hani Jamjoom, and Andrew Warfield. 2013. Split/merge: System support for elastic execution in virtual middleboxes. In *NSDI*. USENIX, 227–240.

[27] Roshan Sedar, Michael Borokhovich, Marco Chiesa, Gianni Antichi, and Stefan Schmid. 2018. Supporting Emerging Applications With Low-Latency Failover in P4. (2018).

[28] Meryem Simsek, Adnan Aijaz, Mischa Dohler, Joachim Sachs, and Gerhard Fettweis. 2016. 5G-enabled tactile internet. *JSAC* 34, 3 (2016), 460–473.

[29] Vibhaalakshmi Sivaraman, Srinivas Narayana, Ori Rottenstreich, S Muthukrishnan, and Jennifer Rexford. 2017. Heavy-hitter detection entirely in the data plane. In *SOSR*. ACM, 164–176.

[30] Manu Sridharan, Stephen J Fink, and Rastislav Bodik. 2007. Thin slicing. In *ACM SIGPLAN Notices*, Vol. 42. ACM, 112–122.

[31] Noa Zilberman, Yury Audzevich, G Adam Covington, and Andrew W Moore. 2014. NetFPGA SUME: Toward 100 Gbps as research commodity. *IEEE micro* 34, 5 (2014), 32–41.