Consistent Composition and Modular Data Plane Programming

Ricardo Parizotto, Lucas Castanheira, Fernanda Bonetti, Anderson Santos, and Alberto Schaeffer-Filho

Emerging programmable data planes enable us to modify switch behavior using software abstractions. However, developing the data plane software is challenging and typically made in a monolithic manner. The authors argue that the data plane should be developed modularly and employ additional abstractions to compose data plane programs and steer packets between them.

Digital Object Identifier:
10.1109/MCOM.001.2000904

Abstract

Emerging programmable data planes enable us to modify switch behavior using software abstractions. However, developing the data plane software is challenging and typically made in a monolithic manner. We argue that the data plane should be developed modularly and employ additional abstractions to compose data plane programs and steer packets between them. This article presents Programming In-Network Modular Extensions (PRIME), a mechanism to compose data plane program modules and define how to steer traffic through these modules. Additionally, the system employs techniques to ensure that updating the steering configuration is consistent according to end-to-end forwarding guarantees. We deployed use cases with existing P4 programs on BMv2, and the results show that PRIME can compose programs with small overheads in terms of latency, the number of forwarding tables, and parser states.

Introduction

Data plane programmability enables the deployment of new features into forwarding devices using software abstractions. Although data plane programmability has promoted greater flexibility in managing and controlling computer networks, this comes at a cost. Programming and configuring the data plane is a challenging task, typically done monolithically for each switch or router. However, the increasing adoption of software-based technologies requires a more dynamic workflow. A promising approach to specifying data plane configurations is developing modular programs using high-level languages (e.g., P4 [1]) and compose them into a single concrete network configuration.

Initial approaches to compose data plane programs used a particular program to emulate many distinct program modules through its table entries [2]. Some improvements in this area focused on reducing the overhead of composed modules [3], and others on finding efficient ways to steer packets through different modules [4]. Despite these improvements, program composition causes overhead on the packet processing. Additionally, updating the steering configuration between modules is error-prone and may create intermediary states, causing misrouting both inside the switch and on end-to-end paths. This problem requires that the composition of data plane programs include mechanisms to ensure transitional consistency guarantees.

Our solution to the problems mentioned above is Programming In-Network Modular Extensions (PRIME), which was introduced in our previous work [5]. As can be seen in Fig. 1, we envision that PRIME could be used by DevOps to systematically specify the composition of P4 modules (e.g., representing different functionalities such as a firewall and a load balancer). Network operators can deploy these composed programs in a host switch and use PRIME to steer packets between composed programs, avoiding that updates create intermediary states. This steering capability would allow a network operator to specify that, for example, the packets should first go through a firewall and then to a load balancer, or vice versa. Further, we provide ways to ensure that the switch configuration remains consistent. In this article, we present a completely revised architecture of PRIME. Different from our earlier work, which was only able to compose programs in a single switch, we now consider an end-to-end update strategy and transitional consistency. We also present new experiments, highlighting the end-to-end approach and showing new use cases with existing P4 programs. We also show a comparison with a state-of-the-art system in terms of parser states, forwarding tables, and throughput.

Background and Related Work

Programmable Data Planes

Programmability enables the deployment of new features into forwarding devices dynamically as software artifacts. One possible approach for modifying the data plane behavior is describing the packet processing using a high-level programming language, such as P4 [1]. In the P4 programming model, packets go through a packet parser that processes user-defined protocols. After the parser processes a packet, it follows a pipeline of control flows, which contains registers, match+action tables, and an apply block that specifies the packet processing. Finally, packet headers are emitted by a deparser or recirculated to the parser.

Related Work

HyPer4 [2] composes data plane programs at a single switch using a virtualization solution. More specifically, the system uses a special P4 program that can emulate many distinct behaviors through

The authors are with Federal University of Rio Grande do Sul.

its table entries. Each composition populates these tables to emulate the original program without rebooting the switch. However, this strategy negatively impacts the performance of the original program with all the overhead of the additional table entries that perform the emulation.

P4Visor [3] proposed the idea of lightweight virtualization of programmable data planes. The system provides multiple operators with different semantics to compose programs to a host program. It performs several optimizations during the merging to reduce the resource consumption of control flows and preserve program isolation. Although techniques to optimize the number of tables between modules (or functions) help reduce resource consumption, P4visor still uses eight tables. Furthermore, P4Visor is conceptually designed for merging a test version to a production version of a program. Consequently, it supports only two compositions at a time and requires modifications to the traffic control to allow more functions to be composed.

Dejavu [4] proposes the use of switch hardware for service function chaining (SFC). The system uses a customized header to index network functions (NFs) and uses recirculation for packets to go through multiple functions. As recirculating packets can generate higher overhead on packet processing, Dejavu programs can divide the same ingress or egress using sequential and parallel operators. These operators reduce recirculations and consequently can allow a higher throughput rate. However, Dejavu is still limited to single switch compositions and would require additional mechanisms to ensure end-to-end compositions.

In this work, we present a system to compose multiple modular data plane programs, considering the ability to perform end-to-end updates consistently in a switch topology. Different from HyPer4, which uses emulation to compose programs dynamically, we choose to perform compositions in an offline mode and dynamically steer flows, similar to P4Visor. P4Visor provides two different testing operators, which compose programs with other constructs to differentiate testing packets from the production version. Unlike P4Visor, we do not need to differentiate testing packets, which reduces the size of constructions necessary for composition. We also describe techniques to steer packets through multiple program compositions, similar to how Dejavu does for chaining functions. Dejavu provides two different composition operators for a single switch, but it does not address how to update the steering configuration consistently. Beyond that, we address the issue of consistent end-to-end updates across multiple switches. Table 1 presents a head-to-head comparison between the main characteristics of PRIME and related work.

PRIME

This section presents PRIME, a mechanism to:
- Compose different PDP programs
- Specify packet steering through program compositions
- Perform end-to-end updates consistently

Overview: Figure 2 illustrates the architecture of PRIME, which is divided into three components: a composition engine, a steering interface, and a consistency checker. First, DevOps perform compositions of modular P4 programs by merging them into one single code. During the composition (Step ①), programs go through a source code analysis to detect and resolve conflicts between program modules. After merging the source code, PRIME exports the steering interface (Step ②) and deploys the new composition, requiring rebooting the switch to instantiate a new functionality. After deploying the data plane functionality, NetOPs use the steering interface to specify which program modules will process traffic in an end-to-end path (Step ③). Traffic is internally routed through programs using the recirculation primitive (Step ④), and a consistency checker running in the control plane ensures that updating the end-to-end steering configuration is consistent (Step ⑤). In the following sections, we describe the techniques employed to implement these components.

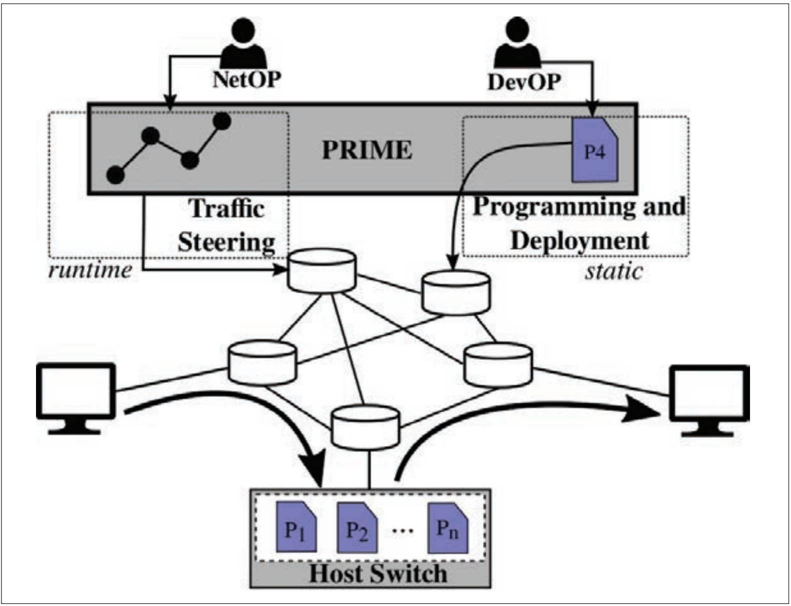


FIGURE 1. PRIME overview.

Work	Multiple operators	Dynamic compositions	Consistent updates	Goals
Hyper4		•		Virtualization
P4Visor	•			Testing
Dejavu	•			Service chainning
PRIME			•	Modular programming

TABLE 1. Comparison between the main characteristics of PRIME and related work.

Programming the PDP

The composition engine is responsible for merging smaller modules into a special program called the host program. The host program has additional building blocks that shape the structure for the program modules' source code. During the composition, we scan the packet parsers and control flows for each module and merge them into the host program if there are no conflicts between programs.

Extending Packet Parsers: The first phase of the composition combines the parser from dif-

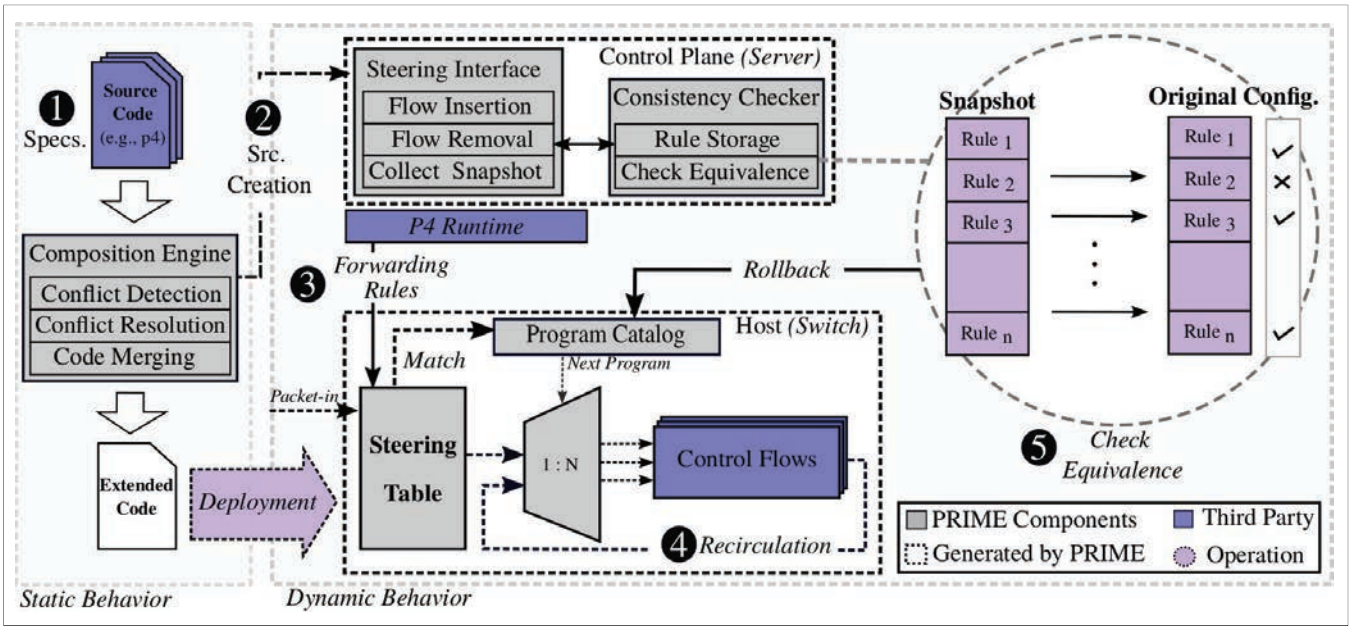


FIGURE 2. PRIME architecture.

ferent modules to the host program parser. This process combines states with the same name and structure and performs the union of transitions. While we process this combination, the parser goes through an analysis phase, which identifies conflicts between different parsers. If the parser does not pass this analysis, PRIME triggers a warning. Otherwise, we can merge the resulting parser into the host program. The new deparser will operate by emitting the headers in an order consistent with the order in which the parser instantiated them. We plan as future work to consider the parser topological graph as part of the composition.

Control Flow Arrangement: After composing parsers and deparsers, we compose the program control flows. Control flows of P4 programs include definitions of match+action tables, stateful registers, and apply blocks. Composing different programs may create conflicts between the definitions of control flow variables. As a consequence, a program could write variables of other composed programs and potentially create conflicting operations. To avoid these conflicts, PRIME identifies equivalent definitions of variables and isolates them by solving ambiguities between their ID and their invocation inside the *apply* block. This process prevents operations over the variables of a program from affecting the state of another program. For example, if we compose a program P_1 that has a table named *table_x*, and a table with the same name is already present in another program, this would be flagged as a conflict. Thus, the P_1 table definition would be rewritten as *table_x_P1*. We also rewrite the primitives that apply this conflicting table, from *table_x.apply()* to *table_x_P1.apply()*, thus preventing P_1 from manipulating the incorrect program table, which could potentially create a mixture of configurations. After solving these conflicts, we can finally place the program source code into the host program structure and deploy the composition into the switch. Further, an additional table, called the steering table, can steer packets for a specific order of modules.

Consistent Steering between Programs

After statically composing all the necessary programs using the technique described earlier, PRIME exports an interface for the P4Runtime, a prototype controller maintained by the P4 consortium, for enabling network operators to specify the steering configuration dynamically. Typical P4 compilers would require writing the P4Runtime methods manually, including the forwarding tables and actions. On the other hand, our technique to export an interface for updating the steering configuration automates this task, shielding both developers and operators from specifying these methods or how the steering occurs.

Steering Configuration: When the NetOP specifies the steering configuration for a specific flow, PRIME stores this configuration in the control plane for further checking the consistency (discussed later). Subsequently, we translate the specification into the steering table entries built up by a host program action called the catalog. The steering table plays a central role in performing the steering. This table intercepts every packet that enters the switch. When a packet matches the table, the catalog loads the steering configuration into user-defined metadata, a per-packet state, and will determine how programs will process this packet.

Ensuring Correctness: After loading the steering configuration, the switch starts processing the programs. After a program module processes the packet, we recirculate it back to the beginning of the pipeline. We also recirculate the steering configuration since the P4 language does not make it automatically. After recirculating, the steering table does not intercept the packet, avoiding it seeing a mixture of two different configurations, which would violate consistency. Subsequently, the switch continues to process the programs in the order specified in the steering configuration. PRIME keeps recirculating the packet until all the programs catalogued by the steering configuration have processed it.

End-to-End Consistency

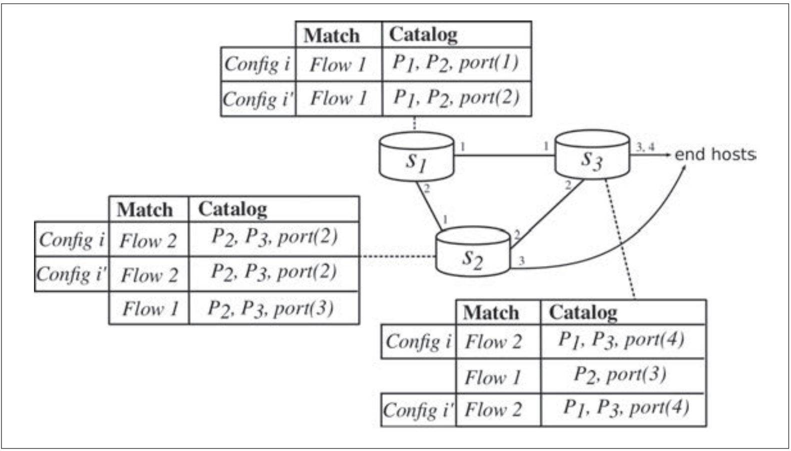
To ensure that an end-to-end configuration of the order of the switch programs is updated consistently throughout the network, one packet cannot see a mix of the old and new steering configuration (as defined in [6]).

Update Strategy: We follow a two-phase update approach [6] to prevent a packet from seeing a mix of different steering configurations. The two-phase update does not require stopping flows to achieve consistency because the switch maintains both the old and new steering configurations during the update. However, flows keep being processed by the old configuration until the control plane finishes sending the new configuration to all switches. Only after that do we enable packets to be processed by the new configuration. This is achieved by using an additional table that tags a packet with a state tag, and only after all switches have the new configuration are the packets tagged with the new configuration tag. Next, the packet is routed through switches, ensuring that the next switch starts processing the packet using the steering configuration corresponding to the state tag within the packet. Figure 3 presents an example of different steering configurations. On Configuration *i*, packets of Flow 1 go through programs $P_1 \rightarrow P_2$ on switch S_1 and program P_2 on switch S_3. On Configuration *i*′, packets of Flow 1 go through $P_1 \rightarrow P_2$ on switch S_1 and follow to programs $P_2 \rightarrow P_3$ on S_2. Considering this scenario, if switch S_1 has updated to the new configuration but not switch S_2 and S_3, the packets of this flow will be marked and processed by Configuration *i* until S_2 and S_3 receive the new configuration. Only after that do we start marking packets with the configuration *i*′ tag. This ensures that the packets will face either configuration *i* or *i*′, but not a mixture of the two, without requiring flows to be stopped.

Checking Consistency: To avoid switch bugs or a malicious attacker rewriting the steering configuration, we draw inspiration from P4Consist [7] to build on a strategy to check for consistency. The consistency checker collects snapshots of the steering configuration and compares them to the control plane rules. When it verifies that a new configuration is on all switches, we commit the change by allowing packets to go through the new configuration. When the consistency checker identifies a rule that differs from what the network administrator specified, it marks it as inconsistent and rolls back the switch state to a previous consistent configuration. In practice, the inconsistent rule will be replaced by the correct one, and PRIME sends a warning to the network operator for analysis.

End-to-End Consistency Case Study

To investigate the benefits of our composition strategy and our techniques to end-to-end consistency, we provide a case study that reproduces the scenario depicted in Fig. 3. The case study uses the same three switches presented in the figure. We also used the same three programs, but instead of naming them generically as P_1, P_2, and P_3, we used a monitoring mechanism designed for security [8], a telemetry system [9], and an offloaded Paxos coordinator [10], respectively. Paxos is a network service that implements the traditional consensus protocol using network hardware. The Paxos coordinator is responsible for receiving client requests and trying to make acceptors agree on them.



	Match	Catalog
Config i	Flow 1	P_1, P_2, port(1)
Config i′	Flow 1	P_1, P_2, port(2)

	Match	Catalog
Config i	Flow 2	P_2, P_3, port(2)
Config i′	Flow 2	P_2, P_3, port(2)
	Flow 1	P_2, P_3, port(3)

	Match	Catalog
Config i	Flow 2	P_1, P_3, port(4)
	Flow 1	P_2, port(3)
Config i′	Flow 2	P_1, P_3, port(4)

FIGURE 3. Steering configuration transition.

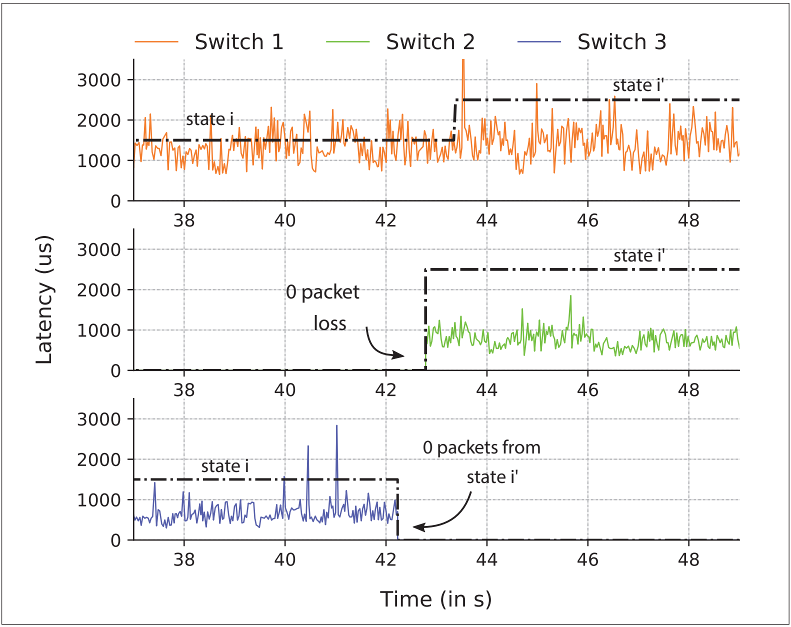


FIGURE 4. End-to-end configuration example.

We show a visualization of the measured latency and highlight the configuration tags for the traffic of Flow 1 in Fig. 4. To measure the latency, we store the difference between the final and initial processing times of this flow into registers and later collect it for analysis. The plot has been annotated to indicate which configuration (whether *i* or *i*′) is currently being used in each switch to process packets of the flow. We obtained this by collecting the configuration tag of each packet. Initially, 608 packets go through configuration *i* on switch S_1, which is composed of the monitoring (P_1) and telemetry (P_2) programs. Next, the same amount of packets goes through S_3, where they are only processed by the telemetry program (P_2). Eventually, assuming that the link $S_1 \rightarrow S_3$ becomes congested, the network operator wants to redirect Flow 1 through $S_1 \rightarrow S_2$ and start processing Flow 1 by the coordinator (P3).

Therefore, what can be seen in Fig. 4 is a transition from configuration *i* to *i*′ that changes the steering of Flow 1 nearly at the 43rd sec-

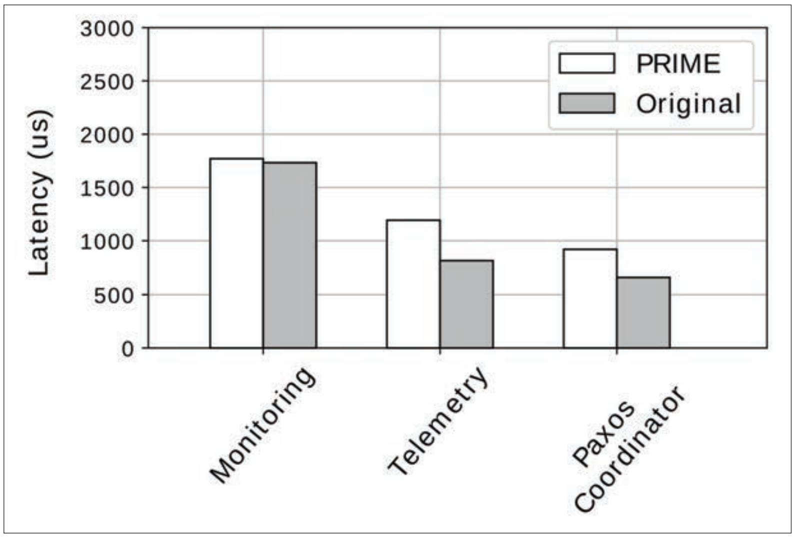


FIGURE 5. Latency × programs.

ond. After completing the transition, 839 packets traverse configuration i' composed by the monitoring (P_1) and telemetry (P_2) programs at S_1. Next, the same 839 packets go through the telemetry (P_2) and the offloaded version of the Paxos (P_3) coordinator at S_2. Suppose that the transition presented in the case study is inconsistent. If it were inconsistent, either we would see packets in S_3 after the commit, or a packet would see configuration i in S_1 after the commit, or S_2 would have dropped Flow 1 packets (since S_2 had no rule for Flow 1 in the previous configuration i and the default action is to drop packets). However, none of these are true in our case study. Therefore, PRIME performed the transition consistently. We want to clarify that in Fig. 4, the transition time in S_1 is slightly different from the time when S_2 starts processing traffic because switches do not have synchronized clocks. However, this does not compromise consistency (i.e., the number of packets processed before and after the transition is in conformance with the consistency notion). Synchronizing data plane clocks is an open research challenge that can be leveraged in the future to better visualize our experiments [11, 12].

Experimental Evaluation

This section presents experiments that quantify the cost of composing programs using PRIME.

Use Case

We composed existing P4 programs with our host program and deployed on the behavioral model (BMv2). We performed 10,000 requests for each composition and gathered switch timestamps to calculate the latency, as in [13]. Since this set of experiments focus on analyzing the composition of programs in a single switch, we configured a topology consisting of two hosts connected to a single switch. We traversed a synthetic workload that triggered packets in an interval of 1 s from one host to another. The experiments were performed on a Linux virtual machine with 2 CPU cores at 2.00 GHz and 2 GB of RAM.

As in the end-to-end consistency case study, we make use of a monitoring mechanism [8], a telemetry system [9], and the Paxos coordinator [10]. We compose these programs with a host program and compare the latency with the original version.

Figure 5 presents the latency that each composition imposes in the data plane. Latency increases compared to the original monolithic version. For example, when PRIME steers packets through the coordinator, the average latency is nearly 200 μs higher than the original program. The same happens with throughput, where the original program reached nearly 3 Mb/s more than the composed version. This occurs because the insertion of additional states to the parser and steering primitives increase CPU consumption. Despite this small overhead, we argue that this is acceptable because our solution has two main advantages: first, it allows multiple programs to share the same switch resources; second, it enables modular compositions, making programming and managing easier.

Comparison with the State of the Art

In our evaluation, we compare our approach (PRIME) to P4Visor [3]. As mentioned in the related work, P4Visor has goals that are different from ours. The system provides test operators that compose different versions of P4 programs. However, by composing programs using P4Visor's *differential* testing operator, we can implement the composition with similar properties as we do with PRIME. Thus, we performed two different compositions using this operator: two instances of a simple router (a production and a test version); and also the router program combined with an alternative implementation of LetItFlow [14]. LetItFlow balances traffic using the concept of *flowlets* (bursts of packets within a flow) and forwards them on random paths.

Code Metrics: First, we compare the number of parser states created by each composition. PRIME requires three parser states in the first composition, while P4Visor requires five (because P4Visor makes additional states specifically intended to parse test packets). Similarly, the second composition requires six states with P4Visor and only four with our approach. Next, we compare the total number of forwarding tables for each composition. For the first composition, P4Visor uses 12 tables, and PRIME uses 7 tables; for the second composition, P4Visor uses 14 tables, and our approach uses only 10 tables.

System Throughput: Similar to our system, P4Visor composes programs into a P4 base program with control constructs to steer packets internally. Once every composition is merged into the host program, its latency will always sum to the latency of the compositions. To be able to compare P4Visor with our approach, we translated the P4Visor base program to P4v16. The translation was required to support the same measurement methodology for both systems and perform a more reliable comparison. We performed an experiment that traversed 1000 packets through the programs with no table entries, that is, we only assessed the host program forwarding structure during the experiment. P4Visor achieves about 8 Mb/s, while PRIME achieves about 12 Mb/s. The result can be explained because PRIME requires fewer lookup operations and parser states than the P4Visor host program.

Concluding Remarks

We propose PRIME, a composition mechanism for P4 programs that also provides ways to steer packets through the programs consistently. We present a functional case study showing a consistent transition between two steering configurations. Experimental results show that ensuring consistency imposes an acceptable impact on latency using a software switch. As future work, we would like to evaluate our proposal on real hardware. Furthermore, we want to explore algorithms that identify dependencies between forwarding tables to update the internal state of program modules [15]. Finally, we want to explore other mechanisms to identify the presence of loops before performing an update.

Acknowledgments

The authors would like to thank CNPq and FAPERGS for the financial support.

References

[1] P. Bosshart *et al.*, "P4: Programming Protocol-Independent Packet Processors," *SIGCOMM Comp. Commun. Rev.*, July 2014.

[2] D. Hancock and J. van der Merwe, "Hyper4: Using p4 to Virtualize the Programmable Data Plane," *Proc. 12th Int'l. Conf. Emerging Networking EXperiments and Technologies*, ser. CoNEXT '16, 2016.

[3] P. Zheng, T. A. Benson, and C. Hu, "Building and Testing Modular Programs for Programmable Data Planes," *IEEE JSAC*, vol. 38, no. 7, 2020, pp. 1432–47.

[4] D. Wu *et al.*, "Accelerated Service Chaining on a Single Switch ASIC," *Proc. 18th ACM Wksp. Hot Topics in Networks*, ser. HotNets '19, 2019, pp. 141–49.

[5] R. Parizotto *et al.*, "Prime: Programming In-Network Modular Extensions," *Proc. IEEE NOMS 2020*, 2020, pp. 1–9.

[6] M. Reitblatt *et al.*, "Abstractions for Network Update," *Proc. ACM SIGCOMM 2012 Conf. Applications, Technologies, Architectures, and Protocols for Comp. Commun.*, 2012.

[7] A. Shukla *et al.*, "P4consist: Towards Consistent p4 SDNS," *IEEE JSAC*, vol. 38, no. 7, 2020, pp. 1293–1307.

[8] L. Castanheira, R. Parizotto, and A. Schaeffer-Filho, "Flowstalker: Comprehensive Traffic Flow Monitoring on the Data Plane Using p4," *Proc. 2019 IEEE ICC*, 2019.

[9] C. Kim *et al.*, "In-Band Network Telemetry via Programmable Dataplanes," *ACM SIGCOMM*, 2015.

[10] H. T. Dang *et al.*, "P4xos: Consensus as a Network Service," *IEEE/ACM Trans. Networking*, 2020.

[11] N. Yaseen, J. Sonchack, and V. Liu, "Synchronized Network Snapshots," *Proc. 2018 Conf. ACM Special Interest Group on Data Commun.*, 2018, pp. 402–16.

[12] P. G. Kannan, R. Joshi, and M. C. Chan, "Precise Time-Synchronization in the Data-Plane Using Programmable Switching ASICs," ser. SOSR '19. ACM, 2019, pp. 8–20; https://doi.org/10.1145/3314148.3314353.

[13] H. T. Dang *et al.*, "Whippersnapper: A p4 Language Benchmark Suite," *Proc. Symp. SDN Research*, ser. SOSR '17, 2017, pp. 95–101.

[14] E. Vanini *et al.*, "Let it Flow: Resilient Asymmetric Load Balancing With Flowlet Switching," *Proc. 14th USENIX Symp, Networked Systems Design and Implementation*, 2017.

[15] M. He *et al.*, "Toward Consistent State Management of Adaptive Programmable Networks Based on p4," ser. NEAT'19, ACM, 2019, p. 29–35.

Biographies

Ricardo Parizotto is a Ph.D. student at Federal University of Rio Grande do Sul (UFRGS), Brazil. His current research focuses on programmable networks and in-network computing.

Lucas Castanheira is an M.Sc. student at UFRGS. His current research interests are distributed systems, programmable networks, and in-network computing.

Fernanda Bonetti is an M.Sc. student at UFRGS. Her current research interests are network function virtualization, containerization, and resource management.

Anderson Santos is a Ph.D. student at UFRGSl. His current interests are network security, formal verification, and network resilience.

Alberto Schaeffer-Filho holds a Ph.D. in computer science (Imperial College London, 2009) and is an associate professor at UFRGS. His areas of expertise are network management, programmable networks, and network resilience. He has authored over 70 papers in leading peer-reviewed journals and conferences related to these topics. He served as TPC Co-Chair for IFIP/IEEE IM 2021, General Chair for SBRC 2019, and Symposium Co-Chair for IEEE ICC 2018 CQRM.

As future work, we would like to evaluate our proposal on real hardware. Furthermore, we want to explore algorithms that identify dependencies between forwarding tables to update the internal state of program modules. Finally, we want to explore other mechanisms to identify the presence of loops before performing an update.