Consistent Composition of Data Plane Programs

Ricardo Parizotto
Advisor: Alberto Schaeffer-Filho

[1]Instituto de Informática
Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre – RS – Brazil

{rparizotto, alberto}@inf.ufrgs.br

Abstract. *Programmable Data Planes (PDP) enable more flexibility in the operation of networks. To fully reap the benefits of programmability, it should be feasible to compose and operate multiple PDP functions into a single target switch as needed. However, existing techniques are not suitable because they lack abstractions for steering packets through the control flows. As such, they do not support the modular composition of PDP programs. This work summarizes the thesis called "Consistent Code Composition and Modular Data Plane Programming" that proposes PRIME, a composition mechanism of in-network functions that also addresses the fundamental needs of packet steering between PDP program modules. We present a design of PRIME, along with use cases. The results show that it is possible to achieve module-wide compositions at little additional cost in throughput.*

Resumo. *Plano de Dados Programáveis (PDP) permitem maior flexibilidade ao operar redes de computadores. Nós argumentamos que, para obter todos os benefícios da programabilidade, deve ser viável compor e operar várias funções em um único switch. No entanto, as técnicas existentes não são adequadas pois carecem de abstrações para o direcionamento de pacotes através das funcionalidades compostas. Portanto, elas não suportam a composição modular de programas para o PDP. Este trabalho sumariza a dissertação chamada "Consistent Code Composition and Modular Data Plane Programming" que propõe o PRIME, um mecanismo de composição de funções na rede que também aborda as necessidades fundamentais de direcionamento de pacotes entre módulos. Apresentamos o design do PRIME, juntamente com casos de uso. Os resultados mostram que é possível obter composições de módulos com pouco custo adicional em termos de vazão.*

1. Introduction

Software-based paradigms for networking enable decoupling software solutions from the hardware they execute, making the management and operation of the network infrastructure more flexible and adaptive. *Software-Defined Networking* (SDN) promotes the separation of the control logic from the forwarding behavior of network devices. More recently, *Programmable Data Planes* (PDP) offer more flexibility in developing protocols and network functionality while allowing packet processing at the line rate. This motivated many emerging applications to bring part of the processing back to the data plane to achieve economies of scale and lower operating costs. As such, operators can leverage programmable hardware to, for instance, process or analyze data, thereby enabling faster reactions in contrast to packet mirroring to middleboxes or controller-based applications.

1.1. Problem Statement

Rather than writing monolithic functions, it should be straightforward for PDP software to be shared and composed into switches as needed. However, existing languages for data plane programming do not support modular development. P4 ("Programming Protocol-independent Packet Processors") [Bosshart et al. 2014], one of the most popular languages for PDPs, requires developers to perform extensive modifications into the function source code to deploy it on existing applications. For instance, if a network operator wants to install a new program in a switch that is already running a P4 program, both programs would require modifications. As a result, researchers have responded with composition approaches that can allocate multiple PDP functions into the same physical target [Hancock and van der Merwe 2016]. The composition typically refers to code merging techniques or virtualization techniques, which can be utilized as a programming model or for the automation of development. Hence, composition tries to avoid rewriting code from different functions manually and maintains the system semantics.

Unfortunately, while composing multiple functions may promote better usage of network resources, the management becomes more complex and error-prone. Current efforts to compose various programs in a single target switch make use of an excessive number of flow tables and parser states [Hancock and van der Merwe 2016, Zhang et al. 2017, Zheng et al. 2018]. Consequently, these techniques can severely limit throughput and increase latency in general-purpose hardware or do not fit in specialized hardware, such as NetFPGAs or ASICs. Moreover, state-of-the-art techniques do not suffice to provide transitional consistency between steering configurations. Without transitional consistency, changes in the steering of flows through the program modules can create intermediary states, which may cause misrouting and security holes [Reitblatt et al. 2012]. Thus, new techniques are required to allow new applications to be composed, preserving transitional packet consistency of traffic steering, without degrading the performance of the data plane operation.

1.2. Research Goals

Composition of Programs: We need to provide a data plane composition mechanism, allowing operators to use the mechanism in different scenarios. The composition must allow different data plane programs to share the same switch resources, promoting better resource usage than monolithic functions on sequences of separate switches. The process of sharing resources must provide intuitions about source code merging to improve resource utilization.

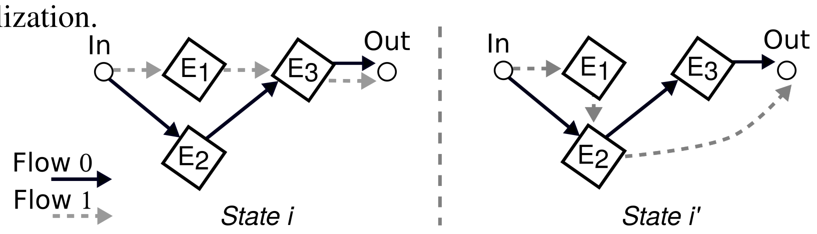

Figure 1. Switch state transition

Steering Definition: Composing multiple P4 programs into devices brings together the necessity of abstractions to steer flows through the composed modules. This, in turn, creates new difficulties for the network operation. We must provide ways to steer packets internally between composed programs. The steering configuration must be easy to manage and semantically coherent with the policy specified by the network operator [Han et al. 2015], i.e., we must support dynamic steering of internal functions avoiding that each update creates state configurations with intermediary states. For example, Figure 1 presents two different states of traffic steering. In the example, network state i steers Flow 1 packets through programs $(E1, E3)$ and Flow 0 through programs $(E2, E3)$, respectively. For some reason, it might be desirable to achieve a transition between the state configuration i to state i', in which $(E1, E2)$ process Flow 1. However, this change of configuration is error-prone and can create undesirable intermediary states, *i.e.,* a packet may see part of state *i* and part of state *i'*. In the example, an intermediary state can be created by performing the update of $E1$ before updating $E2$, leading a new packet to reach $E2$ without having the proper instructions to process it.

1.3. Summary of Contributions

The main contributions of this thesis [Parizotto 2020] can be summarized as follows: *(i)* We design a system for the composition of data plane programs, called PRIME, which also deals with consistent updates of the steering configuration; *(ii)* We implement a case study based on the `V1Model` architecture of our system and evaluate its performance using the behavior model (BMv2) software switch also comparing with a state-of-the-art approach. Our results demonstrate that it is possible to compose multiple data plane programs at a single switch without imposing significant overhead for packet processing.

2. PRIME: Programming In-network Modular Extensions

Figure 2 presents the high-level architecture of PRIME. Firstly, network operators write separated and independent programs, running independently from each other (Figure 2, Step 1). Secondly, the source code of multiple programs is merged into a host program (Figure 2, Step 2), which provides *primitives* to steer packets through them. The composition performs an analysis of the packet parsers and control flows to ensure the program will operate with no loops or ambiguous states. If the merged code passes this analysis, the new program is deployed on the switch (Figure 2, Step 3). Network operators may define which sequences of programs will process a flow dynamically by using a steering interface. The interface updates the switch state to multiplex packets to a specific set of program modules (functions) (Figure 2, Step 4). Specifically, the steering interface produces switch table entries and installs them on the switch (Figure 2, Step 5).

2.1. Programming the PDP

The *composition engine* is responsible for merging smaller modules into a special program called the *host program*. The first phase of the composition combines the parser from different modules to the host program parser. This process combines states with the same name and structure and performs the union of transitions. Next, 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 that the parser instantiated them.

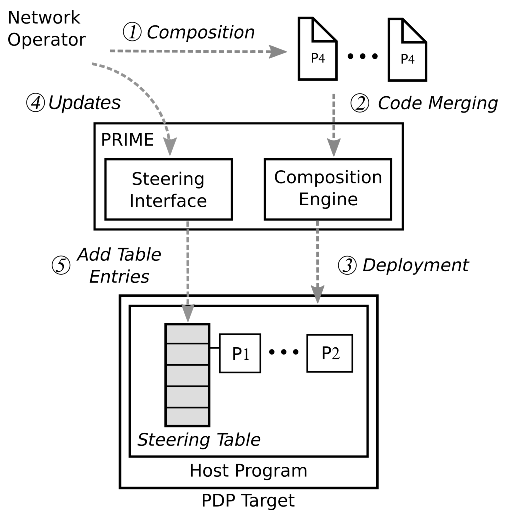

Figure 2. High-Level Architecture of PRIME.

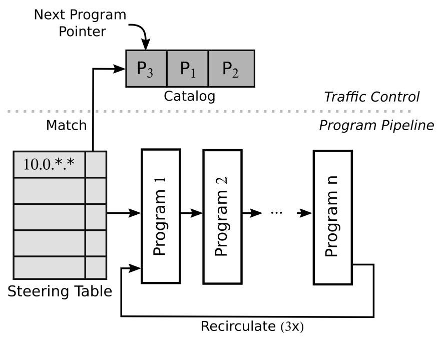

Figure 3. Traffic steering through program modules

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. 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.

2.2. Consistent Steering between Programs

In an update transition, the switch forwarding is updated to a new behavior for a specific flow. We represent an update as a partial function from local packets to a list of programs. To apply an update, PRIME then translates the code to the tuple of parameters of the steering table. When an incoming packet matches the table, an action that we call 'catalog' loads the parameters supplied by the administrator to the internal state. Subsequently, these user-supplied parameters will be stored as packet metadata and used by the host program to determine the order in which program modules are processed.

Figure 3 presents an example of how the steering table can map flows to sequences of programs. In the example, packets that match `10.0.*.*` are mapped to be processed by programs P_3, P_1 and P_2 respectively. For this, after matching the table, the catalog points to the ingress control flow of P_3 and follows to its egress. Next, the packet recirculates and follows to P_1 ingress and egress. Finally, the packet recirculates a third time to P_2. It is important to note that the same data plane structure supports the execution in a different order if the network administrator wishes.

3. Experimental Evaluation

To validate the feasibility of PRIME, we composed existing P4 programs with our host program and deployed on the behavioral model (BMv2). We performed ten thousand requests for each composition and gathered switch timestamps to calculate the latency, as in [Dang et al. 2017]. 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 1sec from one host to another. The experiments were performed on a Linux virtual machine with 2 CPU cores at 2.00GHz and 2GB of RAM.

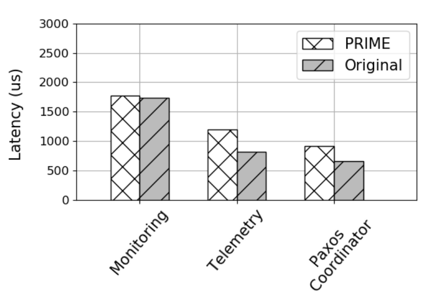

Figure 4. Latency

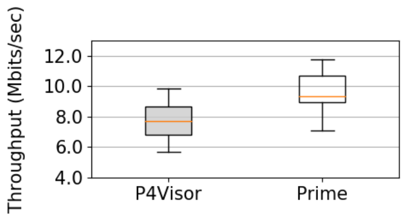

Figure 5. P4Visor vs PRIME: Throughput

We make use of a monitoring mechanism [Castanheira et al. 2019], a telemetry system [Kim et al. 2015], and the Paxos coordinator [Dang et al. 2020]. We compose these programs with a host program and compare the latency with the original version. Figure 4 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 3Mbits/sec 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 management easier.

Comparison with the State-of-the-Art: We also compare PRIME with one of the state-of-the-art approaches, P4Visor [Zheng et al. 2018], to compose programs. P4Visor is a system which compose two different versions of a program using source code merging. Specifically, we utilized the Differential testing Operator of P4Visor to compose programs. We could not build the case studies we presented earlier because P4Visor does not currently support the composition of more than two programs. Thus we show two simple scenarios: a production version of a router with a testing version of the same program, and LetFlow with the simple router program.

Similarly to PRIME, P4Visor composes programs into a P4 base program which has control structures to steer packets internally. In this section, we show how both host programs impact on the latency of packets. Since every composition will be merged to the host, the latency of the host will always sum to the latency of compositions. To compare P4Visor with our program, we had to translate the P4Visor base program to P4v16. The translation was required to support the same measurement methodology to both systems. We performed an experiment that traversed a thousand packets through the programs with no table entries, i.e., we only assessed the standard host program forwarding structure during the experiment. Figure 5 presents the throughput of both base programs. P4Visor achieves 8 Mbits/sec, while PRIME achieves about 11 Mbits/sec.

4. Related Work

HyPer4 [Hancock and van der Merwe 2016] 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 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 [Zheng et al. 2018] 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. Besides that, 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 [Wu et al. 2019] 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 a 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 that uses emulation to compose programs dynamically, we chose 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.

5. Conclusions and Future Work

In this thesis, we presented the design and evaluation of PRIME, a mechanism to help the modular development and composition of P4 programs. PRIME provides techniques to allow new applications to be composed, preserving transitional packet-consistency of traffic steering without degrading the performance of the data plane operation. As future work, we want to perform experiments on real hardware. We also aim to allow dynamic compositions, without requiring to shutdown the device to make a switch composition. Further, in addition to the local guarantees addressed in the composed P4 program, we aim to investigate global path guarantees.

6. Publications

The contributions of this thesis are published (and/or accepted for publication) in the following national and international conferences and journal:

- Abordagem de composição de programas P4 em redes programáveis. In: **XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos (SBRC 2019)** (Qualis: A4) [Parizotto et al. 2019]
- PRIME: Programming In-Network Modular Extensions. In: **2020 IEEE/IFIP Network Operations and Management Symposium (NOMS 2020)**(Qualis: A3) [Parizotto et al. 2020a]
- ShadowFS: Speeding-up Data Plane Monitoring and Telemetry using P4. In: **2020 IEEE International Conference on Communications (ICC 2020)** (Qualis: A1) [Parizotto et al. 2020b]
- Consistent Composition and Modular Data Plane Programming. In: **IEEE Communications Magazine** (Accepted for Publication) (Qualis: A1, Impact Factor: 11.052) [Parizotto et al. 2021]

Moreover, the efforts made during the development of this thesis also contributed to the following publications:

- Flowstalker: Comprehensive traffic flow monitoring on the data plane using P4. In: **2019 IEEE International Conference on Communications (ICC 2019)** (Qualis: A1) [Castanheira et al. 2019]
- A Bottom-Up Approach for Extracting Network Intents. In: **International Conference on Advanced Information Networking and Applications (AINA 2020)** (Qualis: A2) [Ribeiro et al. 2020]

References

Bosshart, P., Daly, D., Gibb, G., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G., and Walker, D. (2014). P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, 44(3):87–95.

Castanheira, L., Parizotto, R., and Schaeffer-Filho, A. (2019). Flowstalker: Comprehensive traffic flow monitoring on the data plane using p4. In *2019 IEEE International Conference on Communications (ICC)*, pages 1–6. IEEE.

Dang, H. T., Bressana, P., Wang, H., Lee, K. S., Zilberman, N., Weatherspoon, H., Canini, M., Pedone, F., and Soulé, R. (2020). P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking*, 28(4):1726–1738.

Dang, H. T., Wang, H., Jepsen, T., Brebner, G., Kim, C., Rexford, J., Soulé, R., and Weatherspoon, H. (2017). Whippersnapper: A p4 language benchmark suite. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 95–101, New York, NY, USA. ACM.

Han, J. H., Mundkur, P., Rotsos, C., Antichi, G., Dave, N., Moore, A. W., and Neumann, P. G. (2015). Blueswitch: Enabling provably consistent configuration of network switches. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 17–27. IEEE.

Hancock, D. and van der Merwe, J. (2016). 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. ACM.

Kim, C., Sivaraman, A., Katta, N., Bas, A., Dixit, A., and Wobker, L. J. (2015). In-band network telemetry via programmable dataplanes. In *ACM SIGCOMM*.

Parizotto, R. (2020). Consistent code composition and modular data plane programming. Master's thesis, Universidade Federal do Rio Grande do Sul.

Parizotto, R., Castanheira, L., Bonetti, F., Santos, A., and Schaeffer-Filho, A. (2020a). Prime: Programming in-network modular extensions. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–9.

Parizotto, R., Castanheira, L., Bonetti, F., Santos, A., and Schaeffer-Filho, A. (2021). Consistent composition and modular data plane programming. *IEEE Communications Magazine*. (To appear).

Parizotto, R., Castanheira, L., Ribeiro, R. H., Zembruzki, L., Jacobs, A. S., Granville, L. Z., and Schaeffer-Filho, A. (2020b). Shadowfs: Speeding-up data plane monitoring and telemetry using p4. In *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, pages 1–6.

Parizotto, R., Castanheira, L. B., and Schaeffer-Filho, A. E. (2019). Abordagem de composição de programas p4 em redes programáveis. In *Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 1028–1041. SBC.

Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., and Walker, D. (2012). Abstractions for network update. In *Proceedings of the ACM SIGCOMM 2012 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '12, pages 323–334, New York, NY, USA. ACM.

Ribeiro, R. H., Jacobs, A. S., Parizotto, R., Zembruzki, L., Schaeffer-Filho, A. E., and Granville, L. Z. (2020). A bottom-up approach for extracting network intents. In *International Conference on Advanced Information Networking and Applications*, pages 858–870. Springer.

Wu, D., Chen, A., Ng, T. S. E., Wang, G., and Wang, H. (2019). Accelerated service chaining on a single switch asic. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, pages 141–149, New York, NY, USA. ACM.

Zhang, C., Bi, J., Zhou, Y., Dogar, A. B., and Wu, J. (2017). Mpvisor: A modular programmable data plane hypervisor. In *Proceedings of the Symposium on SDN Research*, SOSR '17, pages 179–180, New York, NY, USA. ACM.

Zheng, P., Benson, T., and Hu, C. (2018). P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 98–111, New York, NY, USA. ACM.