Building and Testing Modular Programs for Programmable Data Planes

Peng Zheng, Theophilus A. Benson, and Chengchen Hu, *Member, IEEE,*

***Abstract*—Programmable data planes, PDPs, enable an unprecedented level of flexibility and have emerged as a promising alternative to existing data planes. Despite the rapid development and prototyping cycles that PDPs promote, the existing PDP ecosystem lacks appropriate abstractions and algorithms to support these rapid testing and deployment life-cycles. In this paper, we propose P4Visor, a lightweight virtualization abstraction that provides testing primitives as a first-order citizen of the PDP ecosystem. P4Visor can efficiently support multiple PDP programs through a combination of compiler optimizations and program analysis-based algorithms. P4Visor's algorithm improves over state-of-the-art techniques by significantly reducing the resource overheads associated with embedding numerous versions of a PDP program into hardware. To demonstrate the efficiency and viability of P4Visor, we implemented and evaluated P4Visor on both a software switch and an FPGA-based hardware switch using fourteen of different PDP programs. Our results demonstrate that P4Visor introduces minimal overheads and is one order of magnitude more efficient than existing PDPs primitives for concurrently supporting multiple programs.**

***Index Terms*—Programmable Data Plane, Code Merge, Testing, Fault Tolerance**

I. Introduction

PROGRAMMABLE data planes [1]–[3] (PDPs), e.g., Tofino [2], have emerged as a promising alternative to traditional data planes. These PDPs enable an unprecedented level of flexibility: they provide abstractions and language frameworks that simplify the development of stateful network functionality that operates at line rate. This flexibility enables rapid development and prototyping of novel functionality and use cases.

Despite these rapid development and prototyping cycles, the existing PDP ecosystem lacks appropriate primitives and algorithms to support rapid testing and deployment life-cycles. At a high level, many testing paradigms [4]–[6], e.g., canary testing used in Google's [7], [8] networks, require running new versions of a program alongside stable versions. Traffic is split across all versions and the output is compared. Orthogonally, supporting agile development requires composing and merging modular programs together. Furthermore, to enhance system reliability, the widely used techniques are dual-system hot backups for both hardware and software [9]. The backup version runs standby the master program, configured with the latest stable states, being ready to take over once the master program goes wrong. Yet today's PDPs do not provide appropriate virtualization primitives to support multiple versions.

The key challenges to enable these techniques in today's PDP networks lie in efficiently supporting multiple PDP programs and providing flexible operators for the broad range of potential paradigms and state management. Hardware PDP devices include limited physical resources which restrict the size of the PDP programs that can be supported, and enabling multiple versions of a PDP programs on a resource constraint device requires effective algorithms for minimizing resource footprints. Additionally, PDP language abstractions, e.g., P4, provide a limited set of primitives, e.g., P4 does not support loops, and the language restrictions complicate the process of developing general primitives to support a broad range of scenarios. Another challenge is how to provide efficiency state management primitive between multiple PDP programs to enable high network reliability and availability. Although PDPs enable stateful programs, there are insufficient primitives for managing the state. For example, updating the program within a PDP requires rebooting and hence results in unwanted downtime. Few techniques exist to support check-pointing and restoring state or updating PDP state – both of which are crucial components of the development cycles. Specifically, in this paper, we focus on one of the most popular and promising data plane programming languages – P4.[1]

In this paper, we present P4Visor, an abstraction layer and composition primitives, which addresses the above challenges to make testing and development primitives first-order citizens of the PDP ecosystem. The key insight behind P4Visor is that the different versions of a P4 program will share significant code fragments (i.e., tables, parse graph states and action primitives) and thus we can reduce the resource overheads by merging the P4 programs and thus eliminating redundancy. In this way, an administrator can run multiple P4 programs concurrently in the data plane. To further track the state management problem, we provide well-defined primitives to operators, which allow the system record its states periodically to the backup version during the normal operation and also fast rollback when the master program fails.

P4Visor achieves this through a combination of program analysis to identify potential program overlaps and

Manuscript received April 15, 2019; revised February 2, 2020; accepted March 2. The work is supported in part by the NSFC (61672425), the National Science Foundation (through grants CNS-1749785 and CNS-1819109) and China Scholarship Council. This paper was presented the 14th International Conference on emerging Networking EXperiments and Technologies (ACM CoNEXT'18), Heraklion, Greece, December 5, 2018. (Corresponding author: Theophilus A. Benson.)

P. Zheng and C. Hu are with the Department of Computer Science and Technology, Xi'an Jiaotong University, Xi'an, China. E-mail: (zeepean@gmail.com, huc@ieee.org).

T. A. Benson is with the Computer Science Department at Brown University, Providence, RI 02192, USA. E-mail: (tab@cs.brown.edu)

[1]The PDP Programs in the following sections refer to P4 programs unless otherwise stated.

compiler optimizations to merge the P4 programs and reduce resource footprints. To flexibly support different testing paradigms, P4Visor includes domain-specific operators that provide building blocks for composing new testing and deployment paradigms.

Today, the prevalent approach for supporting multiple P4 programs is to virtualize the data plane [10]–[12], e.g., Hyper4 [10], HyperVDP [11], [12], and host different programs atop the virtualization layer. Unfortunately, these approaches [10], [12] require significant resources and are often slow or unscalable [10] because they provide *Full-Virtualization* which uses software to emulate hardware. Rather than providing virtualization and modularity primitives at the software layer, we aim to provide these primitives at the compilation layer which enables us to explore tradeoffs between flexibility and efficiency. In particular, our design choices allow us to trade off a modest amount of flexibility for a significant increase in efficiency.

Logically, P4Visor operates between the PDP programs and the PDP hardware devices, providing merge capabilities to the PDP programs and resource management between the programs. It provides virtualization primitives required for supporting concurrent testing in production networks. P4Visor's goals include security isolation between the management functionality provided by P4Visor's interfaces and data plane functions running on the PDPs devices; efficient resource utilization and management; and, flexible support over arbitrary PDPs targets. To summarize, we make the following contributions:

- **Virtualization Abstractions:** We provide an abstraction for seamlessly merging multiple P4 programs to tackle the resource management and indirection challenges that arise from merging and composing programs (Sec. III).
- **Merge Algorithm:** We present a first look at the code-merging problem for P4 programs. We build a model to theoretically identify the key issues and complexity behind merging P4 programs, and we propose a heuristic to solve it effectively (Sec. IV & Sec. V).
- **Composition Operators and State Managements:** We introduce several composition operators for merging P4 programs to support a range of testing paradigms (Sec. III-C). We provide state management primitives allowing seamless network checkpoint and rollback among different P4 program versions (Sec. III-E).
- **Prototype Implementation and Evaluation:** We implement a prototype of P4Visor's framework and merging algorithms. Using this prototype, we demonstrate the flexibility and efficiency of P4Visor by testing it across multiple P4 programs (Sec. VI & Sec. VII).

II. Motivation

In this section, we describe several well-understood principles used within production networks (*e.g.*, Google and Facebook) to ensure highly-available networks (Sec. II-A), and present a new PDP primitive for effectively supporting these principles in PDP devices (Sec. II-B and Sec. II-C).

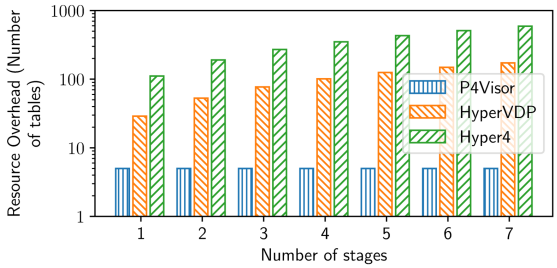


Fig. 1: Comparison of resource overheads for three PDP virtualization approaches under different number of the stages in pipeline.

A. Rapid Development in Large Networks

We briefly describe several techniques which large-scale networking infrastructures employ to ensure that their networks remain highly-available in the face of changes.

Canary Testing (A-B Testing [5], [6], [13]). Canary testing, well documented by Google's [6]–[8] and Facebook's [5] networking and infrastructure teams, requires running multiple versions of a program alongside each other. Canarying (or A-B Testing) tests new code by sending a subset of traffic through the code (*e.g.*, 1% of traffic) and, if nothing "bad" happens, slowly increases the subset of traffic using the test code until all traffic is using the test code.

Fault Tolerance (Data-Diversity [4], [14]). To improve security and availability, certain networks run multiple instances of their control plane, perturb the instances with some randomness, and then compare the outputs from these versions. The system uses the most popular output. This approach directly tackles bugs and overcomes intruders. Facebook [14] runs four control planes and compares the output between these control planes.

High Availability (Network-Rollback [15], [16]). While testing techniques exist to eliminate bugs, unfortunately, bugs still occur in production networks [8], [17], [18]. To tackle this, most companies, e.g., Google [7], [8], store checkpoints of stable version and rollback to the previous version of the code. This use case requires maintaining several versions (or snapshots) and managing network states such as efficiently and consistently swapping between them.

Modular Code. Extensive work in the software engineering community [19] and recently validated by large software engineering firms (e.g., Facebook [5], [17]) have demonstrated that the key to successfully supporting rapid prototyping and deployment of complex functionality is modularity (code-reuse). Yet, today programmers are forced to write monolithic P4 programs. Missing from the ecosystem is a framework for effectively supporting multiple modular PDP Programs (similar to processes in operating system) and composing them together.

B. Novel PDP Primitives: Code Merge

Today, the most direct approach for supporting the aforementioned techniques is to use virtualization, *e.g.*, HyperVDP and Hyper4. Unfortunately, HyperVDP and Hyper4 incur significant performance and resource overheads. In Fig. 1, we present the memory overheads of using these different virtualization techniques with an emphasis on the number of

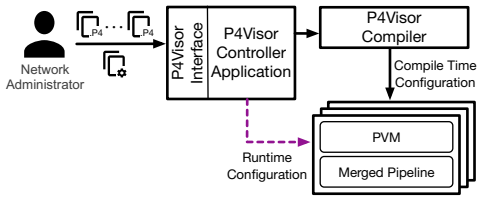


Fig. 2: P4Visor workflow: the key components and how they work together.

tables used. The overheads grow linearly with the size of the program because both techniques declare a fixed number of additional tables to emulate each of the P4 program's stages and primitive actions – their hypervisors have to use these tables to record the runtime states for each program. For example, to run a P4 program with two pipeline stages, HyperVDP and Hyper4 have to declare at least 53 and 191 tables respectively which limits the number of primitive actions supported to 9 and prevents HyperVDP and Hyper4 from supporting more sophisticated programs such as Switch.P4 [20] which has 19 primitive actions.

Motivated by the inefficiencies of existing virtualization primitives, in this paper we aim to answer the following question: *Is it possible to have a framework for supporting multiple versions without incurring the overheads of full virtualization?*

To answer this question, we investigate the design of a lightweight virtualization based on a source code merging primitive. The merge primitive takes as input N P4 programs and creates as output one P4 program that combines all input P4 programs but retains the functionality of each of the original P4 programs. As an example shown in Fig. 3, our new primitive takes, as input, the abstract representations of two P4 programs (the programs P_1 and P_2 in Fig. 3) and combines them into one (Fig. 3 (a)). Central to providing this primitive is ensuring that during the merge, P4-specific correctness constraints (*e.g.*, table dependencies) are maintained while efficiency is maximized through resource reuse.

The merged P4 program should give each P4 program the illusion of sole occupancy on the hardware. Our approach differs from full virtualization in several ways: first, while full virtualization provides virtualization through a special P4 program, we provide virtualization through the P4 program compiler. Our approach offers one key benefit: whereas full virtualization needs to allocate resources to support any potential P4 program, we only need to allocate resources to support the P4 programs being compiled. This specialization minimizes the number of additional tables required to support the combined program. Second, full virtualization does not explicitly support modularity and composition of multiple P4 programs into one, whereas, we can directly support these use cases.

C. P4Visor Workflow

Next, we present the workflow of P4Visor to illustrate P4Visor's components and how they work together. To support the envisioned testing paradigms, the network operators must provide P4Visor with (1) the different P4 programs to merge, (2) the type of testing composition operators to implement

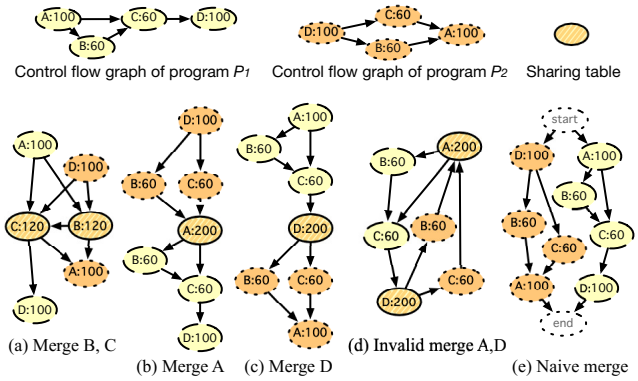


Fig. 3: Illustrates various approaches for merging of two P4 programs. (a) demonstrates an intelligent merge with two share tables; (b) and (c) are two possible merges with one shared table; (d) is an invalid merge; and (e) demonstrates a simple combination of two programs which doubles the resources.

(*e.g.*, A-B testing or Differential testing) – the composition operator determines the policy for splitting and comparing traffic, (3) the amount of traffic used for testing, e.g., test X% of the traffic, and (4) the traffic sampling granularity, e.g., all packets of a flow should be consistently tested, test any packet of any flow, or just test flows within a specific subnet. Network administrators configure these settings using either a simple command line or a configuration file. Given such a testing specification, P4Visor installs code at edge switches to consistently tag packets for testing and to remove the tags before the packets exit the network. Tagging at the edge enables P4Visor to ensure that the different switches consistently test the same packets and that we can perform end-to-end tests across the whole network. During the merge, P4Visor adds tables to compare the output from the different program and generate packets to the controller when these results differ. These packets allow operators to reason about the implications of the new code. In Fig. 2, we present P4Visor's workflow.

To support this workflow, P4Visor requires (1) a domain-specific configuration language for configuring the testing paradigms (the P4Visor interface), (2) a merge algorithm (discussed in Sec. IV and Sec. V), (3) a framework for implementing the merge and supporting the indirection required to support the merge (discussed in Sec. III), (4) techniques for enabling comparisons and techniques for tagging/untagging the packets.

III. Design of P4Visor

In this section, we first provide an overview of P4Visor's architecture (Sec. III-A) and the detail of P4Visor's compiler design (Sec. III-B). Next we introduce the currently supported composition operators (Sec. III-C & III-D) and state managements primitives (Sec. III-E).

A. Overview

In Fig. 4, we present the architecture for P4Visor. At a high-level, P4Visor is composed of four components: the P4Visor Interface (PVI), P4Visor Compiler (PVC), P4Visor controller Application (PVA), and P4Visor Management agent (PVM).

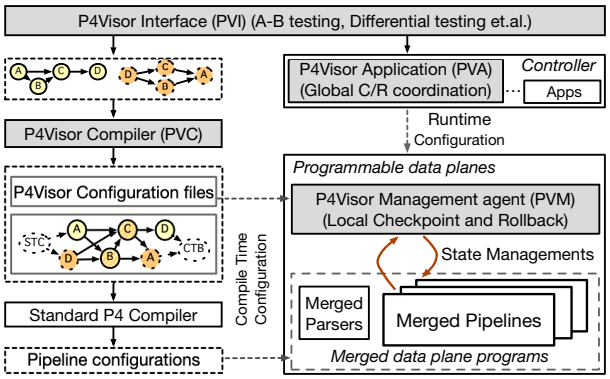


Fig. 4: P4Visor overview. The left part presents the procedures of the compile phase while the right part outlines architecture of P4Visor for runtime phase.

P4Visor Interface (PVI): PVI runs on a server and provides the management interface for the network administrator (or developers) to use to control the composition of different P4 programs. In our current prototype, we implemented two operators: *A-B Testing* and *Differential Testing* (described in Sec. III-C).

P4Visor Compiler (PVC): PVC takes, as input the P4 programs, from the PVI, and returns, as output, a merged P4 program and P4Visor-specific configuration files which provide a mapping between the resources of each input P4 program and the merged P4 programs. The PVC analyzes the parse graphs, tables and control flows of the input P4 programs and merges them. The key to the PVC is the P4 program-merge algorithm (Sec. IV & Sec. V), which identifies the data plane resources within all input P4 programs that can be "safely" merged while maintaining the semantics and dependencies of each P4 program.

P4Visor controller Application (PVA): PVA runs on the controller with a global view of the network, providing runtime control over the testing operators. For example, in A-B Testing, the PVA populates the testing traffic control tables (Sec. III-D) for all the edge switches, identifies testing traffic.

P4Visor Management agent (PVM): PVM runs on the PDP devices, *i.e.*, programmable switch, intercepts messages between the control plane (controllers) and the merged P4 Program and uses the P4Visor-specific configuration files produced by PVC to determine how to appropriately modify the messages. Essentially, the agent multiplexes and demultiplexes messages between the different controllers and the merged P4 program. What's more, PVM also provides state management to the different P4 programs running on the local switch, such as the state checkpoint and rollback under the coordination of the controllers (Sec. III-E).

B. P4Visor Compiler

P4Visor merges PDP programs by merging their parse graphs and their control flow graphs.

Merging Parse Graphs: A naive approach for merging the parse graphs of two programs is to emulate the merge process by resubmitting packets into the pipeline multiple times; once for each of the states in both parge graphs (this is the approach taken by Hyper4 [10]). Unfortunately, this incurs a significant performance overhead — each time a packet is recirculated, throughput is cut and packet processing latency is increased.

Instead, we merge the two parse graphs into one and use a tag (*i.e.*, TFlag) to disambiguate and break conflicts in the merged parse graph. Fig. 5 presents an example of two parse graphs being merged. Recall, each parse graph is a finite state machine (FSM) with each state representing the bit offsets of each header type. In merging these parse graphs, we align the parse graphs' FSMs and merge identical states.[2] In Fig. 5 (c), the merged states are in orange with solid line. We observe that Ethernet and IPv4 are both identical and thus can be merged. There is ambiguity about when to parse the VLAN and the IPv6 headers, and we break this ambiguity by introducing the TFlag state: packets with the TFlag, *i.e.*, test packets, should parse the IPv6 header type, whereas only packets without the TFlag, *i.e.*, non-test packets, should parse the VLAN header type. Note: we need only insert one such state for the TFlag, and this state can disambiguate all potential ambiguities in all merged states.

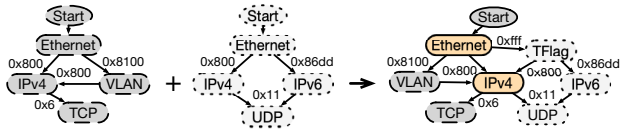
Merging Control Flow Graphs: P4Visor analyzes the pipelines of all P4 programs to be merged and identifies the tables to be merged using the algorithms presented in Sec. IV. Given this information, P4Visor merges the P4 programs by: 1) rewriting the Table IDs – to avoid conflicting IDs [3], 2) rewriting "GoTo" statements for all tables except the merged tables to reflect the new Table IDs, 3) for merged tables, P4Visor does one of two things: if the merged table leads to one table, then rewriting is obviously just rewriting the existing "GoTo" to use the appropriate ID; However, if a merged table leads to more than one table, e.g., table "B" or "C" in Fig. 3 (a), then P4Visor will add multiple "GoTo" statements, one for each branch.

Recall the above example, shown in Fig. 3, in the control flow graph of program P_1 the next-hop for "C" is "D", in the merged graph, table "C" will retain "D" as its default next-hop table modulo rewriting IDs to reflect D's new ID; however, P4Visor will also add a "GoTo" that matches on the TFlag for the other control program P_2 and uses "A" as the next-hop for packets with the TFlag tag.

As a result of the merge process, P4Visor creates a new P4 program, which is a normal P4 program that can be run through a standard PDP compiler. Additionally, P4Visor creates a P4Visor-specific file, called the P4VisorConfiguration, which provides a mapping between the resources of each input P4 program and the merged P4 programs. Recall, each P4 program uses unique IDs to identify tables, and during the merge, these IDs may be modified. The P4VisorConfiguration provides a mapping between the original ID and the modified IDs: this file allows P4Visor to transparently rewrite all calls between the control and data plane that use these IDs. For example, when the controller sends a flow entry installation message to the switch to add a new entry to a *resource-*

[2]Since we focus on merging different versions of the same program, and we anticipate a high degree of overlap between the parse graphs of the different programs.

[3]While each program may have unique Table IDs, multiple programs may reuse the same Table IDs which will create problems during compilations



(a) Production parser (b) Testing parser (c) Merged parser

Fig. 5: Merging two parse graphs.

sharing table, PVM will modify the table IDs according to the P4VisorConfiguration to ensure that the entry is installed correctly.

Besides, P4Visor compiler also analysis the stateful objects in both the production and testing programs and generate another P4Visor-specific file file, called P4VisorStatesConfigure. This file contains the state size and map information in the merged P4 programs. Note that when different P4 Program can have the stateful objects of the same type or name, P4Visor assigns unique identifies to each of the stateful objects in the merged program.

Preserving Traffic Isolation: Our framework must be able to offer isolation whenever it is required. Wherein, we define isolation as the following property: if two PDP Programs, P_1, and P_2, are isolated, then traffic for P_1 is never processed by resources exclusively dedicated to P_2. Additionally, table entries controlled by one are never modified by the other. To enable isolation, we introduce an ACL-Bit (attached to tables) that provides access control overflow entries in the resource-sharing tables.

Observe that each table in the original P4 programs will map to exactly one table in the merged P4 program, while each table in the merged P4 program may correspond to one or more tables. We label all tables in the merged P4 Program that correspond to multiple tables in the original programs as *resource-sharing* tables. In Fig. 3 (a), nodes C and B are both *resource-sharing* tables. For the *resource-sharing* tables, the P4Visor compiler will add the ACL-Bit to the table entries to provide traffic isolation for P4 programs. Combined with the TFlag which identify the packet as test packet, this ACL-Bit provides traffic isolation by allowing packets to match entries in the shared table only when packets match both the TFlag and the ACL-Bit. While the TFlag and ACL-Bit ensure isolation within resource-sharing tables, the TFlag alone ensures isolation between non-resource-sharing tables.

In addition to isolating the packet processing, the ACL-Bit also enables P4Visor to separate control over entries in the flow tables: The control plane for a P4 program can only modify the entries with the appropriate ACL-Bit value. Given this ACL-Bit, each P4 program can update the shared table correctly without side effects to the other P4 programs.

C. Composition Operators

To illustrate the flexibility of our merge algorithm, we use it to implement two distinct testing composition operators.

A-B Testing Operator: This operator allows multiple programs to run side by side in a production network with a subset of traffic siphoned to the testing version, as shown in Fig. 6 (a). To support, our A-B Testing composition, P4Visor must securely and flexibly manage traffic among multiple versions. To ensure security, *i.e.*, production traffic will not be processed by the testing programs, P4Visor adds/removes a special flag (TFlag) to packets at edge switches when traffic enters and leaves the network.

Differential Testing Operator: The key difference between the A-B Testing and Differential Testing operators is that: while the A-B Testing operator is mutually exclusive, *i.e.*, traffic either goes to the production or the test P4 program, for the Differential Testing operator, the test packets must be copied and send through both programs, with outputs compared at the end of the pipeline. The packet life-cycle is shown in Fig. 6 (b).

D. Primitives for Composition Operators

To support these two operators, P4Visor must provide a flexible primitive for controlling traffic and, specifically, for Differential Testing, P4Visor must provide primitives for performing comparisons.

Flexible Control: To ensure flexibility, a traffic management module, called Testing Traffic Control (TTC in Fig. 6), is developed and inserted into the merged pipeline to identify a packet as either "test" or "production" traffic and guide the packets along the appropriate pipeline. As shown in Fig. 6, the TTC module is instantiated within the first table that all packets encounter and affixes the TFlag header to the packet once the packet is identified as the testing packet.

The TTC contains a set of stateful registers and flow tables, which are configurable using the P4Visor Interface. Using the P4Visor interface, network operators can configure the TTC to configure how traffic is sampled for testing:

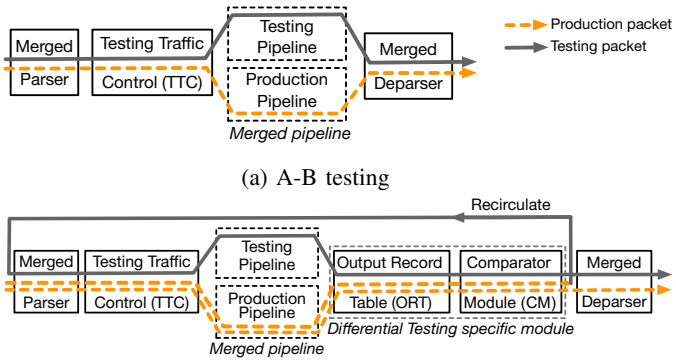
- *Random sampling:* Operators can specify which percentage of traffic is randomly selected and piped through the testing "pipeline" and which percentage of traffic goes through the production "pipeline", *e.g.*, randomly sample 1% of the total traffic for testing.
- *Flow based sampling:* Alternatively, operators can specify the exact flows that should be sampled by specifying a flowspect, *e.g.*, traffic from subnet "10.10.10.0/24" should be sampled.

The Comparison Primitive: To enable comparisons, a special table, called an output record table, is added at the end of each program's pipelines. This table's fields are configurable through the P4Visor interface. Specifically, we need to clone the packets and, in turn, compare their outputs at each switch.

To clone packets, we leverage the recirculate primitive, which recirculates a packet and allows the packet to be processed multiple times by the switch. We recirculate once for each version we want to test – during recirculation, we also recirculate the metadata fields. At the end of the pipeline, the packet is processed by the Comparator Module (the CM in Fig. 6), which compares the output of the different versions. The comparator reports to the controller, via a message, when the compared packets and metadata fields are different. [4]

Using the P4Visor Interface, a network administrator can 1) specify which outputs should be compared; 2) configure how to process the differences detected by the Comparator. In

[4]To control the overheads associated with recirculating packets, the operators can fine-tune the number of packets sampled to ensure the system provides acceptable performance.



(a) A-B testing

(b) Differential testing

Fig. 6: Life-cycle of a packet in P4Visor (under our two composition operators).

general, the Comparator can support two kinds of comparisons either on packet header fields or on metadata fields.

E. Primitives for State Management

To further take advantage of P4Visor's merge algorithm, we next present the design of state management for multiple PDP programs.

Recall that the widely used technique is running multiple instances of the program to enable high reliability and availability, which is now directly supported within programmable networks by P4Visor. In this case, with dual versions of the PDP program running side by side, one is the master version, and the other is standby version. The key to achieve high fault tolerance is providing the agile state migration among multiple PDP programs versions to minimize state migration time upon failures. While many existing works provide algorithms to ensure consistency of checkpoint [16] and updates [21], [22], our goal is to provide performance guarantee for state operation within each data plane device, which is crucial yet missing in the PDP system. To this end, we implemented two state management primitives, *i.e.*, the checkpoint and rollback for P4Visor.

Checkpoint: Checkpoint primitive migrate the states from the master version to standby version according to the P4VisorStatesConfigure file, as shown in Fig. 7. So that the standby version stores the latest correct checkpoint of the master, which is ready to take over the master version whenever the master failures are detected. Note that in this paper, we focus on the network states that are configured by the controllers, which can be divided into two types in PDP programs: the first is the rules in tables; the second one is the stateful resisters configured to control the traffic processing, such as the proportion of random sampling in some programs or the meter states to limit the rate.

Rollback: Rollback primitive allows the network administrator rollback the current state back to the last stable snapshot, which is stored in another version running standby. To do this, P4Visor simply guide the traffic from the master version to the standby version. In this way, P4Visor allows network rollback with minimal side effect, *e.g.*, rebooting or unwanted downtime. It requires tagging the packets to ensure the packet level consistency during the network rollback or update [21],

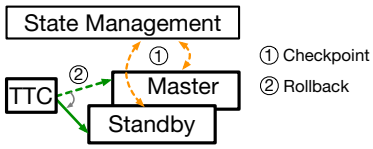


Fig. 7: State management primitives: checkpoint and rollback.

while our traffic isolation module, *i.e.* the TTC in Fig. 7, directly ensure this consistency. P4Visor provide predictable operation time because it doesn't require any state operation during the rollback, while existing works often have to analysis state to compute how to update the state which cause unpredictable performance [16]. Instead, P4Visor has the last correct configuration ready in standby program with minimal delay for rollback operation.

IV. Merging P4 Programs

In this section, we provide an overview of PDP-based resource constraints (Sec. IV-A), present the design of P4Visor's source code merge model (Sec. IV-B), and conclude by theoretically analyzing the complexity and hardness of the problem.

A. Background on P4 Compiler Constraints

In general, there are two kinds of constraints on a P4 program. These constraints are either placed on the compiler by the language (hardware target independent) or placed on the compiler by the hardware architecture (target-dependent). An example of a target-independent constraint is that there can be no loops in the control flow graph; hence, it needs to be a directed acyclic graph (DAG).This constraint is invariant across all targets. However, the target-dependent constraints vary dramatically from target to target and are especially hard to enforce without intimate knowledge of the target hardware's proprietary details. For example, RMT [23] has 32 stages in its pipeline while Intel's FlexPipe [1] has 5 stages with different memory constraints for each stage.

To tackle these two constraints, P4 compilers are split into two components: a target-independent compiler (front-end compiler) and a target-dependent compiler (back-end compiler). In this paper, we focus on the design of target-independent merge optimizations. We aim to, first, provide a general optimization that benefits all hardware-targets. Our target independent optimization builds on the insight that merging different tables results in significant savings across all hardware targets for multiple reasons: merging tables reduces overheads associated with instantiating tables and merging tables results in large tables which take advantage of various hardware optimizations (we elaborate on this in Sec. VII).

B. Merging Optimization

Merging two P4 Programs is fundamentally equivalent to merging two weighted DAGs into a single weighted DAG with the added objective of minimizing space (*i.e.*, the number of nodes). To the best of our knowledge, no existing work has explored this problem: specifically, merging two weighted DAGs into one while maximizing overlap. The most closely related works [24], [25] provide suboptimal results, we elaborate on them in Sec. VIII. Next, we more formally describe the problem.

We model a program's control flow using a Table Dependency Graph (TDG) [26] $G = (T, E)$ where vertices

$T = \{t_1, t_2, ..., t_n\}$ [5] and edges $E = \{(t_i, t_j) \mid t_i, t_j \in T\}$ map to the tables and the table dependency, respectively. Each table $t_i \in T$ has three attributes:

1) the program id, $t_i.pid$, reflects the P4 program in which the table resides;
2) table ID, $t_i.tid$, reflects the table's ID and helps to differentiate tables; and,
3) table size, $t_i.size$, reflects the memory footprint of the table (size is a function of width and number of entries defined).

Given G, we can compute the dependency matrix, D, of the graph as: $D[t_i, t_j] = 1$ if there is a dependency path from t_i to t_j and $D[t_i, t_j] = 0$ otherwise [6].

For simplicity, we formalize the PDP-merge problem for two P4 programs, but the problem formulation and analysis generalizes to cases with more than two P4 programs.

Objective: Our goal is to merge two programs – a production version denoted as $G_r = (T_r, E_r)$ with the dependency matrix D_r and a testing version $G_s = (T_s, E_s)$ with the dependency matrix D_s into a single program $G_m = (T_m, E_m)$ with the dependency matrix D_m, while minimizing the total resources required. In this paper, we only focus on table memory resources. Restated, our object is maximizing sharing resources:

$$max \sum_{i=1}^{|T_m|} w_i \tag{1}$$

where w_i is the weighted contribution of reducing the resources in G_m used by table $t_i \in T_m$.

We define the set of *resource-sharing* tables, T_{ms}, as a subset of tables in the merged TDG G_m: these tables in T_{ms} are merged from multiple tables in the original programs, which satisfy the following constraints: equivalence, correctness, and loop-freedom.

For each table $v_i \in T_{ms}$, w_i captures both the memory type and table size. Currently, the memory size is calculated as a function of the number of entries and the width of each entry:

$$w_i = c_i \cdot len_i \cdot width_i$$

where len_i and $width_i$ are the number of entries and width of an entry in table t_i respectively. c_i is a configurable weighted coefficient that allows an administrator to guide our optimization algorithm to merge tables that the administrator cares about. For example, if an administrator only cares about the TCAM tables, she can set the table weights of all non-TCAM types to 0. As a preprocessing step, P4Visor sets $w_i = 0$ for each table $v_i \notin T_{ms}$ which shares no table resources with other tables because these tables cannot be merged. Note that when the weights for all tables T_{ms} are equal, the objective function (1) leads to a merged TDG that minimizes the total number of tables.

Target-Independent Constraints: Two tables, $t_{r_i} \in T_r$ and $t_{s_j} \in T_s$, can be merged if and only if three constraints are satisfied:

(i) *Equivalence:* The two tables are structurally equivalent (same actions and match fields but they can vary in the number of declared entries). Here the equivalent tables are assigned the same id, that is $t_{r_i}.tid = t_{s_j}.tid$.

(ii) *Correctness:* the table dependencies of both tables are maintained – correctness is preserved.

$$\begin{cases} D_m[t_{r_i}, t_{r_j}] = D_r[t_{r_i}, t_{r_j}], \ \forall t_{r_i}, t_{r_j} \in T_r \\ D_m[t_{s_i}, t_{s_j}] = D_s[t_{s_i}, t_{s_j}], \ \forall t_{s_i}, t_{s_j} \in T_s \end{cases} \tag{2}$$

(iii) *Loop-free:* the resulting graph is loop free, that is, the dependency matrix of G_m satisfies $\forall t_i, t_j \in T_m$,

$$D_m[t_i, t_j] \cdot D_m[t_j, t_i] = 0 \tag{3}$$

Target-Dependent Constraints: While this work focuses on target-independent constraints, here, we briefly sketch out how target-dependent constraints can be introduced into our problem formulation.

Abstractly, we can introduce target-dependent constraints by introducing hardware information. One constraint placed by hardware is the number of physical stages. For example, RMT [23] has 32 stages, and thus RMT can only support P4 programs whose crucial dependency path length is no more than 32. To overcome this limitation, we can add a constraint that limits the merged TDG's critical path length to less than 32. This may force our algorithm to explore solutions that create merged programs that do not maximize overlaps, but that ensure shorter critical dependency paths.

We proved that our TDG merging problem can be reduced to and from the Maximum Weighted Independent Set (MWIS) problem: a problem which has been proven to be NP-Complete [27]. Due to space limitation, more details on complexity analysis and proof of the problem can be found in Appendix A-C.

V. Efficiency

Next, we design a heuristic to efficiently solve the problem in real time (Sec. V-A) and discuss a systematic approach for configuring resource sharing of entries within the merged tables (Sec. V-B).

A. P4Visor Heuristic Merging

A naive approach for solving the "merge" problem is to perform a brute-force search through all potential combinations in G_m to find the solution which provides the maximum overlap: the best-known optimal algorithm for solving the maximum independent set problem is Bron-Kerbosch. We implemented the extended Bron-Kerbosch [28] and observed that it can only handle small graphs and was unable to scale to large graphs (i.e., greater than 80 nodes): In particular, given a 7-day time limit, we were unable to solve the Bron-Kerbosch algorithm for graphs with over 80 nodes. Thus, Bron-Kerbosch was unable to process the largest DAG in our dataset (Switch.P4 which has over 120 tables). Motivated by the inadequacies with Bron-Kerbosch, we designed a new heuristic to solve the merge problem.

Heuristic: Our heuristic is based on simulated annealing (SA) which has proven effective in solving the MWIS problem [29]. In our heuristic, each state of the search space is defined as a subset V_{sub} of the vertex set of graph V_p and every vertex in V_{sub} is nonadjacent to the other vertices. Motivated by prior work [30], [31], our heuristic generates neighboring states to

[5] n is the total number of tables in the pipeline.
[6] $D[t_i, t_i] = 0$ because P4 programs are generally acyclic graphs.

TABLE I: The parameters of the heuristic

Parameter	Default	Range	Description
INIT_T	200	100-1000	The initial temperature. The larger means more diversification for the SA process.
FINAL_T	0.1	0.1-1	The final temperature. The smaller means more intensification for the SA process.
COOLING	0.999	0.99-0.999	The cooling parameter. The lager means slower convergence rate and needs more time.
NO_CHANGE_CNT	250	25-250000	The local optima indicator. The larger means more intensification to reach global optima.
SA_ITER_TIME	200	200-200000	The iteration time. The larger means more randomness and possibility to global optima.

explore using one of the following two procedures: (1) adding one vertex $v_i \in V_p \setminus V_{sub}$ to V_{sub} and deleting all the vertices,v_j in V_{sub}, that are adjacent to v_i. (2) adding two nonadjacent vertices $v_i, v_j \in V_p \setminus V_{sub}$ to V_{sub} and deleting one vertex from V_{sub} that is adjacent to both v_i and v_j. The energy function, or evaluation function, of our heuristic is defined as the total weight, $E(V) = \sum_{v_i \in V} weight(v_i)$, of the vertices in current search state, V, where $weight(v_i)$ is the weight of each vertex $v_i \in V_{sub}$. A new state will be accepted if its energy, *i.e.*, $E(V_{n+1})$, is larger than the current state's energy $E(V_n)$; otherwise we accept the new state with probability of $e^{-\frac{\Delta E}{t}}$. The temperature, t is initially set to INIT_T initial and decreases linearly to FINAL_T at each iteration based on this equation $t(n+1) = t(n) * \texttt{COOLING}$. We terminate the search when the temperature decreases to FINAL_T.

Optimizations: A well known problem of SA-based heuristics is that they can often get stuck in a local optima. To avoid this problem, we employed multiple optimization mechanisms over the simulated annealing search process. First, we use a counter threshold NO_CHANGE_CNT to signal the local optima. During the search, if the heuristics performs a threshold number of iterations without an improvement, then we end the search process and start a new one. As the initial temperature is high in each new SA process and is more likely to accept the solutions worse than the local optimal solutions, more perturbation and randomness are introduced to the following search process. We run the simulated annealing process SA_ITER_TIME times which increases coverage over the search space and introduces more randomness. To provide a good trade-off between the diversification and intensification during the whole search, we dynamically adjust the cooling parameter so that COOLING increases with the growing of the SA iteration times. In this way, a newer SA process has better intensification to reach the global optima during the search. To summarize, the increased randomness and larger coverage over the search space, enables our heuristics to avoid local optimas.

Parameters: Several key parameters help us to seek the right trade-off between the running time overhead and solution quality, as summarized in table I. The first two parameters INIT_T and FINAL_T control the search diversification and intensification for each annealing process. More diversification and intensification is more likely to reach the global optima however requires more running time. COOLING allows us to control the convergence rate of the SA process. The following two parameters NO_CHANGE_CNT and SA_ITER_TIME focus on providing more randomness and possibility to get rid of local optimas.

B. Controlling Merge Space Savings

In general, merging the control flow graphs of two P4 programs consists of two major steps. The first is to identify the tables to be merged in both P4 programs, G_r and G_s, that satisfy the "correctness" constraints. The second step is to merge the control flow according to the identified tables.

In merging two tables, t_{r_i} and t_{s_j}, the resulting table, t_{m_x}, can vary in size, number of entries, ranging from $max(t_{r_i}.size, t_{s_j}.size)$ (the merged tables reuse 100% of their resources) to $t_{r_i}.size + t_{s_j}.size$ (the merged tables do not reuse any resources and the logical size of the merged table is equivalent to the sum of the original tables).

At both extremes, P4Visor provides benefits. At the extreme where no table entries are shared, i.e., $t_{m_x}.size = t_{r_i}.size + t_{s_j}.size$, the merge provides benefits because it enables G_m to fit within smaller memory by reducing the overheads associated with instantiating individual tables in hardware, *e.g.*, in Xilinx's Virtex-7 FPGAs [32] instantiating a TCAM table with 64 bits width × 256 bits depth consumes 2 RAM blocks but instantiating TCAM tables with 4 times (128 bits width × 512 bits depth) or 16 times (256 bits width × 1024 bits depth) more memory consumes only 3 or 5 RAM blocks respectively. Thus, merging would save resources, even if the merged table contains the same number of entries as the original tables combined.

At the other extreme, where switch memory is reused, that is, $t_{m_x}.size = max(t_{r_i}.size, t_{s_j}.size)$, then precious switch memory is being saved by sharing resources across tables. This savings is in addition to the savings of overheads described earlier. At this extreme, the table resources are shared proportionally between the different programs based on the fraction of traffic allocated to each program.

To explore this trade-off, P4Visor exposes a parameter, k, to the operator through the P4Visor Interface. This parameter allows the operator to control the amount of sharing: $k = 1$ means no sharing while a $k = 0$ means proportional sharing.

VI. P4Visor Implementation

We have implemented the P4Visor compiler in 3000+ lines of Python code and 800+ lines of C++ code. The controller application, P4Visor interface and P4Visor management agent are all developed with Python in over 300+ LoCs. The output, of which, can then be fed into a back-end compiler. The state management module is built on the software switch runtime API. The P4Visor compiler takes as input the high-level intermediate representations of P4 programs (i.e., HLIR) and merges them into one program. Merging the high-level IR allows us to operate at a platform independent level while maintaining the complete semantics of the P4 language. Currently, we only support merging of $P4_{14}$ programs. However, our algorithms and framework are also applicable to $P4_{16}$ programs. P4Visor's source code is online in our Github repository [33].

A. Supporting Flexible Testing Operators

We now discuss, in detail, the implementation of several interesting architectural components: specifically, the Differential Testing specific module. In this discussion, we also,

```
1  registers cnt, Rate
2  action sample_testing_pkt() {
3    register_write(cnt, 0, cnt+1);
4    modify_field(testing_meta.testingbit,
5                 cnt%Rate);}
6  table testing_traffic_identify {
7    //fields are configurable using P4Visor
       Interface
8    reads {ipv4.dstAddr : lpm;}
9    actions {sample_testing_pkt;
10            set_testingbit;}}
11 table testing_traffic_control {
12   reads {testing_meta.testingbit : exact;}
13   actions {goto_test_pipe; goto_prod_pipe}}
```

Fig. 8: Code Excerpt from our TTC Implementation.

```
1  //compare the outputs
2  if(testing_meta.meta_p!=testing_meta.meta_t){
3    apply(diff_procedure);
4  }
5  //an example procedure configuration
6  action diff_procedure(testing_meta,
7           mcast_group) {
8    update_fields(testing_tag, testing_meta);
9    set_output_mcg(mcast_group);
10 }
```

Fig. 9: Pseudocode for Comparator.

demonstrate the flexibility of the testing operators provided by P4Visor.

Testing Traffic Control (TTC): In Fig. 8, we present an excerpt for our implementation of the TTC component. Recall, the key goal of the TTC is to provide flexible control over the sampling and selecting of traffic for testing. P4Visor provides both runtime and compile-time control which allows the administrator to alter sampling configuration program: at compile time the administrator can configure aspects of the flow spec to match on and at runtime, the administrator can configure the packets to sample.

The control flow of TTC in the system is implemented in table `testing_traffic_ control` (Lines 11-13), the TTC uses the metadata `testingbit` to determine if the traffic should use the testing pipeline or the production pipeline.

Run Time Configuration: The administrator can control the sampling rate (by changing the registers (Line 1)) and the subnets to be sampled (by modifying the entries in `testing_traffic_identify` table (Line 6)). The sampling frequency is implemented as a special action, `sample_testing_pkt` (in Line 2), which uses two registers, `cnt` and `Rate`, to determine the sampling rate, *e.g.*, sample one packet in every R packets. [7] The subnet sampling is performed by comparing the addresses in the packet against the entries in table `testing_traffic_identify`.

Compile Time Configuration: Additionally, the structure of table `testing_traffic_identify` can be altered, through the PVI, to reconfigure the TTC and allow the TTC to match packets for sampling based on other aspects of the FlowSpec beyond subnet.

Comparator: In Fig. 9, we present an excerpt of our code for the Comparator Module. The Comparator is implemented using a set of flow tables with actions associated and conditional nodes in the pipeline, providing a configurable interface to the network operators. The outputs of each version of the program are recorded to a set of metadata (i.e., *meta_p*) and then compared by Comparator (Line 2) to determine if the versions are different. If a difference is detected, the action `diff_procedure` is used to create a packet to send to the controller. To overcome a limitation of our target platforms, we create a new packet by multicasting the original packet and sending a version to the controller (Lines 6-10).

[7]Due to limited arithmetic operations supported by P4, the TTC can only support the sample ratio of $1/R$ ($R = 1, 2, 3\ldots$).

B. Limitation of Existing PDP Targets

PDPs are expected to provide a rich set of packet processing features, e.g., the action primitives defined in P4. However, current PDP targets, *e.g.*, software switch Bmv2 [34] or FPGA-based hardware from Xilinx SDNet [35], can only support a limited set of P4's features. Several of the key P4 features required to enable P4Visor to include 1) stateful registers; 2) packet cloning, for creating multiple copies of a packet to be processed by different programs; and 3) in-switch packet generator, for generating a packet to the controller that summarizes differences between the two P4 programs.

- While packet clone primitives are defined in the P4 specification, the clone feature is not supported by the Bmv2 target. We address this problem by attaching attributes of the packet to the metadata and recirculating the packet and metadata through the pipeline for processing by the alternative programs. Thus, by recirculating the packet, multiple versions of a PDP Program can independently process the same packet.
- In-switch packet generator is not supported by either the Bmv2 or FPGA targets (Xilinx SDNet). To send the testing outcomes to the controller, P4Visor adds those fields to the pre-configured TFlag and then sends the packet out to the controller.
- Our hardware target is even less flexible than the Bmv2 due to the limitation of the current development toolchain (SDNet). Specifically, SDNet does not support stateful register which impacts our design of the comparator and limits the set of programs we can deploy. To support P4Visor on our FPGA-based hardware target, we have implemented those primitives in low-level hardware (i.e., the Testing Traffic Control module is implemented with 1000+ lines of Verilog code). We believe these hardware limitations will be addressed with the evolution of the SDNet toolchain.

VII. Evaluation

A. Experiment Setup

We have evaluated P4Visor on both a software (Bmv2 [34]) and a hardware (ONetSwitch [36]) programmable data plane.

Software PDP: On the Bmv2 target, we analyze the following programs: Reference Switch.P4 [20], L2 switch, Simple Router, NAT, VLAN and Arp-Proxy, Flowletting [37], and Heavy Hitters [38]. The Bmv2 runs in mininet with a single switch, two hosts for testing, and a third host for running the controller. Before testing, we install flow entries into the tables so that the two end hosts can ping each other. We use iPerf with the default TCP window size to measure the throughput.

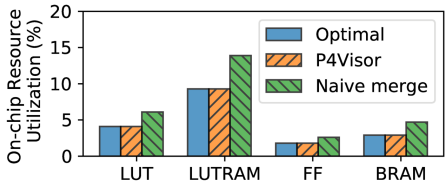


Fig. 10: Impact of program merging approaches on memory utilization.

Hardware PDP: On the ONetSwitch target, we were only able to evaluate the following programs: L2 Switch, Simple Router, and VLAN. We were limited in the set of programs because ONetSwitch builds on Xilinx's Zynq SoC [36] which only supports a subset of P4's language features (see Sec. VI-B). To test the performance of the switch, we connect two PCs with 10G NIC to the ONetSwitch45 switch, due to NIC limitations, the maximum achievable throughput for our servers is 5Gbps. We use iPerf to generate traffic between the hosts and similarly crafted rules to force traffic through as many tables as possible.

B. Performance Benefits and Overheads

Here, we evaluate the overheads of P4Visor and analyze the practical benefits of source code merging as a lightweight virtualization primitive.

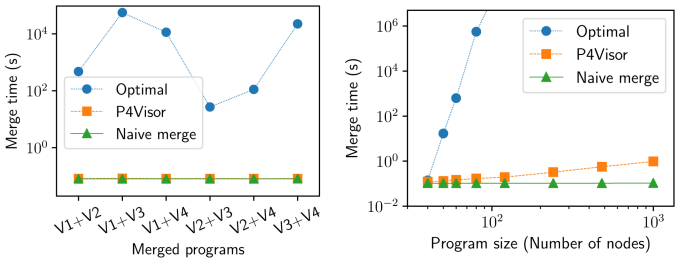
1) Benefits of Resource Sharing

To understand and quantify the benefits of resource sharing, we have compared P4Visor's merge algorithm against a *Naive merge* algorithm [39], which is a greedy algorithm for MWIS problem. When solving the problem, *Naive merge* selects a vertex of minimum degree, removes it and its neighbors from the graph until no vertex available.

Our results show that merging introduces significant benefits for three distinct reasons: First, instantiating a table into hardware incurs some overheads. Thus by having two programs sharing a table, we ameliorate the associated overheads and this translates into memory savings. For example, while multiple tables may use the same actions, these actions need to be independently stored for each table and by merging tables, we reduce the number of instances of these actions. Second, as tables grow in size, several of the resources increase in a sub-linear fashion due to hardware-specific optimizations, *e.g.*, the BRAM memory in Xilinx includes optimizations that result in sub-linear growth. Third, when we modify the parameter k which impacts the amount of sharing, we reduce the total number of entries. This introduces yet more savings.

To illustrate the first two points, we analyze a simple P4 program (the router program [40]) consisting of two tables: a TCAM IPv4 routing table (32 bits width, 256 entries) and an exact match IPv6 routing table (128 bits width, 256 entries); each table has two actions. For this analysis, we set $k = 1$ which means both the P4Visor-merged and the naive-merged programs will have the same number of entries. We merge two router programs into one and then compile the merged programs to the ONetSwitch45. In our analysis, we focus on the four different kinds of memory resources of the Xilinx chipset: LUT, LUTRAM, FF, and BRAM.[8] Fig. 10 illustrates

[8]The LUT, LUTRAM, and BRAM are mainly used to store Table structures — with BRAM used to store large TCAM tables. The FF, is however, mainly used to store timing and control signals.



(a) Real P4 Programs (b) Synthetic P4 Programs

Fig. 11: Runtime of program merging approaches.

our first point – our heuristic merging with P4Visor results in a 32% to 49% savings in resources compared with the naive merging algorithm. Note that our algorithm achieve the same solution as the optimal algorithm in this case, and more evaluation with the optimal algorithm lies in Sec. VII-C. To illustrate the second point, we analyze the amount of resources required to support tables of varying sizes. We observe that while most resources grow linearly with the size of the tables, the BRAM grows sub-linearly (figure omitted due to space limitations).

Takeaway. Intelligently merging tables leads to tremendous resource savings. We anticipate these savings to only grow as P4 programs become even more complex.

2) Performance Overheads

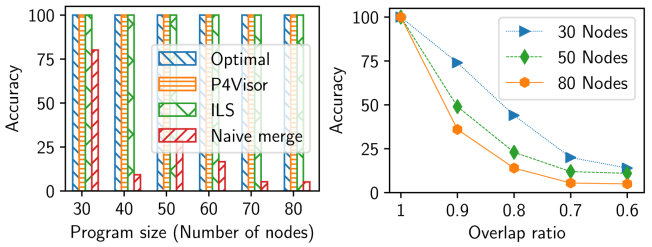
To evaluate the performance overheads of P4Visor, we randomly select two of the evaluated PDP programs, *i.e.*, L2switch, Router, VLAN, NAT, and Flowleting (only the first three for hardware switch), merge them with P4Visor, and compare the throughput/latency of running traffic through the merged program against that of running traffic through the unmodified programs – we compare the merged program against the better performing of the two original PDP programs. Our experiments, not shown due to space constraints, demonstrate that P4Visor introduces minimal overheads. Specifically, the TTC and the Comparator modules which add tags to packets and perform comparisons introduce minimal overheads. In the software switch, the throughput decreases by less than 1.5% and the delay penalty is less than 3%. For the hardware switch, the overhead is much smaller, both the throughput and delay overheads are less than 1%.

Takeaway. P4Visor introduces several tables and actions to support the different testing paradigms; however, these constructs introduce minimal performance overheads to the network (less than 1% in hardware) making them highly desirable for today's networks.

C. Analytical Evaluation of the Heuristic

To evaluate the efficiency and accuracy of our heuristic, we compare our heuristic with the optimal solution (Bron-Kerbosch), a naive greedy merge [39], and a state-of-the-art algorithm for MWIS problem called hybrid iterated local search (ILS) heuristic [31]. Note: we were unable to evaluate the optimal approach on programs with over 80 nodes because the optimal algorithm failed to provide a solution. In these evaluations, we focus on two kinds of P4 programs:

First, real P4 programs, is based on the reference Switch.P4 [20], which contain 82 tables in the ingress pipeline and



(a) Synthetic programs (b) Impact of overlap ratios

Fig. 12: Accuracy of program merging approaches.

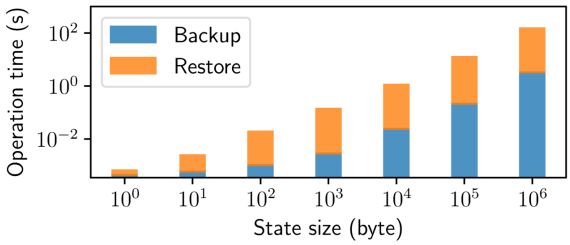


Fig. 13: The operation time with the increase of state size.

41 nodes in the egress pipeline. As Switch.P4 is built in a configurable fashion, we create different versions by turning on or off specific functionality. Specifically, we created the following four versions: (1) Switch.P4-V1: by turning on the OpenFlow processing module. It has 84 ingress nodes and 53 egress nodes. (2) Switch.P4-V2: by turning off the Tunnel processing module. It has 66 ingress nodes and 30 egress nodes. (3) Switch.P4-V3: by turning off the ACL processing module (MAC, IPv4, IPv6, RACL/PBR). It has 76 ingress nodes and 37 egress nodes. (4) Switch.P4-V4: by turning off the Multicast processing module. It has 73 ingress nodes and 39 egress nodes.

Second, synthetic P4 programs, which enable us to systematically evaluate the accuracy and efficiency of our heuristic at scale – larger than the largest known P4 program (Switch.P4). We generate synthetic programs ranging in size from 30 to 1000 tables with randomly generated dependencies (edges) – we generate the edges to ensure that the graph maintains a graph density of 0.4.

Efficiency: To evaluate the efficiency of our heuristic, we measure the time it takes to merge two randomly selected P4 programs. Fig. 11(a) presents the runtimes for merging real P4 programs. We observe that our heuristic and the naive algorithm consistently take a similar amount of time (~0.1 seconds) and both are considerably faster than the optimal algorithm. Fig. 11(b) presents the runtimes for merging synthetic P4 programs: this figure highlights the relationship between the runtime of the three approaches and P4 programs size (in number of nodes): the optimal algorithm shows an exponential growth, while both our heuristic and the greedy approach show a linear growth as a function of program size. With the larger graphs, we observe that while our heuristic performs slower than greedy, it's performance is still acceptable. For ILS heuristic, we focus on the accuracy of the algorithms as the running time largely depend on the parameters such as the iteration times of the search. We tuned the parameters so that the ILS takes similar time with our heuristic.

Accuracy: Next, we analyze the accuracy of the different approaches to understand the cost of the performance improvements. To evaluate the accuracy, we compare the solutions generated by the different approaches against the optimal solution. Within the real programs, we observe that the greedy approach, ILS and our heuristic achieved 100% accuracy; however, for the synthetic programs, as shown in Fig. 12(a), we observe that the greedy approach achieved an average accuracy of 30% while our heuristic was able to achieve 100% in all situations. With the further increase of the program size, our algorithm is able to produce the best solutions given by ILS heuristic in all the cases.

Upon further analysis, we observe that the accuracy of the greedy heuristic is a function of the ratio of overlap between the different programs. Recall, the real programs are all variants of Switch.P4 thus we expect there to be significant overlaps. If the overlap is extremely high, as it is common when minor changes are made, then the greedy heuristic performs well; however if the overlap is low, e.g., when significant changes are made, then accuracy drops. To illustrate this point, in Fig. 12(b) we explore the impact of overlap ratio on accuracy. From this figure, we observe that for the greedy approach, the amount of overlap has a large impact on its accuracy.

Takeaway. While our heuristic is slower than the greedy approach, our heuristic scales linearly and provides better accuracy across a broader set of scenarios and is comparable with the state-of-the-art algorithm. In short, our heuristic is fast and accurate.

D. Use Case 1: Testing P4 Programs

In section VI-A, we have shown how to flexibly configure the fields to be tested and actions to be performed using P4Visor interface. In this section, we demonstrate the use of P4Visor to perform testing and illustrate how these interfaces may be configured. Specifically, we use P4Visor to perform Differential Testing to test the behaviors of two versions of the P4 Router program. Unlike the previous sections which focus on overheads and accuracy, here we explore the operational interactions involved with P4Visor.

Testing Setup: To use P4Visor, we (1) configure P4Visor to record and compare the 32-bits next-hop metadata fields of the programs. To handle the detected differences, (2) configure the Comparator to send the packets along with the outputs from two programs to the controller (the same as the configuration in Fig. 9), and (3) fed the configuration files and the two P4 programs into P4Visor's interface to produce the merged program.

Testing results: We evaluate the merged program on the Bmv2 target. At runtime, we control the routing tables of the two programs with two different routing applications. We observed that the P4Visor application can detect differences, via the P4Visor-generated messages, within milliseconds once the control planes install different routing entries for the same flow. With the help of the outputs stored in the messages, an administrator can further debug and analyze the behavior of the tested programs.

E. Use case 2: Towards Fault-tolerance PDP

Given two versions of PDP programs merged by P4Visor, we run one version as the master program and configure another as redundancy version on permanent standby to enhance the reliability and availability. During the normal operation, PVMs backup the latest correct checkpoint of the master versions to the redundancy versions under the coordination of the controller. Once the master version occurs failure, *e.g.*, due to the program bugs or misconfiguration, the network administrator will rollback to the latest stable as soon as possible. In this case, the redundancy version allows seamless rollback without interrupting the traffic. To evaluate the operation time of our state management primitives, we merge two Flowletting programs [37] and run the merged program on Bmv2 switch. We send the checkpoint and rollback messages to the PMV to trigger state migration between two program versions.

With various state number in programs, the evaluation results show that the rollback operation keeps constant about 0.5ms because the rollback primitive only need to configure the TTC module regardless of state number in programs. However, the checkpoint operation time grows linearly with the number of state, as shown in Fig. 13. During the checkpoint, PVM first read the states from the main version (marked as backup in Fig. 13), and then write back to another version (marked as restore in Fig. 13). We find that restore operation dominates the operation time because the runtime API only allows burst operation for read. While runtime API is built on thrift lib, the agent can get multiple stateful registers in one request for backup; however, for writing back, the agent is only allowed to write one register upon each request. So writing operation takes much more time than reading especially when the state number increasing.

F. Use case 3: Towards end-to-end testing

With the merged program of different versions, P4Visor provides the ability to run testing traffic on one program without side effect on another. Different from the static model or emulate based approach, P4Visor allows us to perform holistic testing scalable to large production topologies with real traffic running on the testing versions.

In this use case, we demonstrate how to use P4Visor to perform end-to-end testing across the network, *e.g.*, the global reachability. Given a testing network in P4Visor, the key to enabling end-to-end testing is to generate appropriate traffic. To do this, we take the network topology as input and generate the packets for each pair of two edge ports. Then at runtime, we send the generated packets to one side of each edge ports pairs and check the output on the other port to test the reachability. Here we also note that automatic test packet generation tools, *e.g.* ATPG [41] can be integrated to improve the efficiency of traffic generation.

VIII. Related Works

The most closely related work [42] explore source code merging as a method for providing virtualization. We explore a similar approach but in the P4 domain and tackle a host of domain-specific issues. Moreover, we prove the complexity of the merge problem. Below we explore related works on SDN composition, DAG-Merging, and other recent work on programmable data planes.

PDP. Many have explored virtualization [43]–[45], update [46]–[48], and state-management [15], [49] techniques for traditional SDNs. These approaches assume a stateless data plane, focus on the control plane and are thus operate at software speeds, which are orders of magnitude slower and less dynamic than the data plane. Despite the growing emergence of PDP-based architectures and solutions, to-date, there are few principled approaches for supporting testing P4 Programs. Most work focus on applications of PDPs [50]–[53] or developing interfaces and primitives to enrich existing PDPs environment [23], [54], [55]. Our work falls in the latter class and argues for a principled extension of PDPs to include interfaces, abstractions, and primitives to enable testing — in short, to support rapid prototyping.

PDP Compiler. Work [26] is the first toward building a general PDP compiler with a target-independent front-end and a target-dependent back-end. They identified key issues, *e.g.* table size, program control flow, and hardware memory restrictions faced by all PDP compilers. Our framework contributes novel primitives such as lightweight virtualization and modularization to the ecosystem by adding more intelligence to the PDP compiler. Several other works [35], [56], [57] have explored challenges associated with compiling P4 programs to various hardware targets, e.g. FPGA [35], [57] or RMT [56]. Our work can benefit from these approaches by using these approaches within the back-end compiler – note that in this paper, we focus our emphasis on front-end optimizations.

PDP Virtualization. Related works [10]–[12] proposed a general virtualization solution for PDP, using a large hypervisors program to support multiple programs. Instead, P4Visor provides lighter weight virtualization primitive at compiler layer based on code merging and achieves one order of magnitude more efficiency than hypervisor based approaches. Another approach called hard virtualization [58]–[60], presented an conceptual hardware architecture to support virtualization for P4-based switches. They are limited to the reconfigurable hardware, *e.g.*, FPGA, while our software based solution is general to support different type of PDP targets without changing the hardware architecture [61] [62].

SDN Composition. Orthogonal work on composition and modularity in SDNs [45], [63]–[66] focus on SDN rules and not on P4 program's source code. Concurrent work on composition [67] aims to support orthogonal composition operators. In this work, we present the first attempt to formalize the problem and present a framework with supporting algorithms and abstractions to enable composition within PDPs.

DAG-Merging. The problem of merging two DAGs has been explored by several others [24], [25], [68]. Works [24], [68] merge processing graphs for network functions with the objective of minimizing the path length, *i.e.*, reducing packet processing latency in the merged processing graph. Our goal differs from those works because we aim to maximize the number of merged nodes, *i.e.*, minimizing the resources used by the merged graph. Saha et al. [25] provide a sub-optimal heuristic for merging two unweighted DAGs into one unweighted DAG with the objective of minimizing the

number of vertices in the final DAG. Our objective varies as we want to simultaneously decrease the number of vertices while maximizing the magnitude of overlap since the vertices in our problem are weighted. More importantly, unlike prior work, we are the first to formalize and prove the complexity of the DAG merge problem, which enables an efficient heuristics based on simulated annealing.

IX. Discussion

Control Plane Overheads. In addition to the data plane overheads, P4Visor introduces overheads to the control plane. For example, the P4Visor agents running on the controller and the switch PDP devices, have to multiplex and demultiplex messages between the controller to the local P4 programs. This translation introduces processing overheads and also memory overheads because the agents need to maintain a mapping and perform the translations. Additionally, re-using the control channel between the controller and switches to transfer packets summarizing the result of the tests reduces the available bandwidth on the control channel. As part of future work, we plan to explore approaches, e.g., SwitchVisor [69], to effectively share and partition these control plane resources.

Target Dependent Optimizations. As discussed in Sec. IV, our current efforts focus on target-independent optimizations (i.e., front-end compiler), as part of future work we will extend our formulation to tackle the back-end compiler by introducing constraints and objectives specific to the hardware targets.

Seamless Reconfiguration. While full virtualization provides support for headless updates (reconfiguring the data plane without a reboot), our approach requires a reboot after every reconfiguration. As part of future work, we plan to tackle issues related to these reboots by intelligently migrating state, e.g., with SwingState [54], and reconfiguring paths, e.g., with zUpdates [70], during the reboot to eliminate disruption.

Composition Operators. This work has focused on supporting testing-specific composition operators; however, as part of on-going work we are exploring composition operators for enabling code modularity, e.g., parallel and sequential composition [63]. Supporting these operators requires extending our current formulations to account for operator specific constraints.

X. Conclusion

In this paper, we propose a lightweight virtualization primitive for testing P4 programs through code merging. To support this primitive, we present a framework, called P4Visor, which uses compiler optimizations and program analysis to achieve efficient source code merging. We evaluate the theoretical complexity of the merging algorithm and present an efficient greedy heuristic. Our work opens up space for implementing a wide range of composition operators and frameworks for PDP programs. We outline several avenues for future exploration. The first is to conduct an extensive evaluation on using P4Visor configuration for automating configuration debugging and network diagnostics. Second, the target-specific optimizations at the back-end of the PDP compiler will further improve the efficiency of PDP programs. At last, we think providing data plane virtualization primitives at the compiler layer is a promising direction considering the domain-specific constraints of PDP.

References

[1] R. Ozdag, "Intel ethernet switch fm6000 series-software defined networking," 2012.
[2] B. Networks, "Barefoot whitepaper: The world's fastest and most programmable networks." 2016. [Online]. Available: https://barefootnetworks.com/resources/worlds-fastest-most-programmable-networks/
[3] I. Cavium, "Xpliant ethernet switch product family." 2018. [Online]. Available: http://www.cavium.com/XPliant-Ethernet-Switch-ProductFamily.html
[4] E. Keller, M. Yu, M. Caesar, and J. Rexford, "Virtually eliminating router bugs," in *5th ACM CoNEXT*, 2009, pp. 13–24.
[5] K. Veeraraghavan, J. Meza, D. Chou *et al.*, "Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services," in *12th USENIX OSDI*, 2016, pp. 635–651.
[6] D. Zhuo, Q. Zhang, X. Yang, and V. Liu, "Canaries in the network," in *15th ACM HotNets*, 2016, pp. 36–42.
[7] A. Singh, J. Ong, A. Agarwal *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," in *ACM SIGCOMM*, 2015, pp. 183–197.
[8] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from googles network infrastructure," in *ACM SIGCOMM*, 2016, pp. 58–72.
[9] "Dual-system hot backup," 2013. [Online]. Available: https://support.huawei.com/enterprise/en/doc/EDOC1000012899?section=j00g
[10] D. Hancock and J. van der Merwe, "Hyper4: Using p4 to virtualize the programmable data plane," in *12th ACM CoNEXT*, 2016, pp. 35–49.
[11] C. Zhang, J. Bi, Y. Zhou, A. B. Dogar, and J. Wu, "Hyperv: A high performance hypervisor for virtualization of the programmable data plane," in *26th IEEE ICCCN*, July 2017, pp. 1–9.
[12] C. Zhang, J. Bi, Y. Zhou, and J. Wu, "Hypervdp: High-performance virtualization of the programmable data plane," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 3, pp. 556–569, March 2019.
[13] R. Alimi, Y. Wang, and Y. R. Yang, "Shadow configuration as a network management primitive," in *ACM SIGCOMM*, 2008, pp. 111–122.
[14] M. Jimenez and H. Kwok, "Building express backbone: Facebook's new long-haul network," 2017. [Online]. Available: https://code.facebook.com/posts/1782709872057497/building-express-backbone-facebook-s-new-long-haul-network
[15] Y. Zhang, N. Beheshti *et al.*, "Netrevert: Rollback recovery in sdn," in *3rd ACM HotSDN*, 2014, pp. 231–232.
[16] Y. Yu, C. Qian, W. Wu, and Y. Zhang, "Netcp: Consistent, non-interruptive and efficient checkpointing and rollback of sdn," in *26th IEEE/ACM IWQoS*, June 2018, pp. 1–10.
[17] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng, "Robotron: Top-down network management at facebook scale," in *ACM SIGCOMM*, 2016, pp. 426–439.
[18] A. G. Martinez, *Chaos Monkeys: Obscene Fortune and Random Failure in Silicon Valley*, 2016.
[19] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, pp. 1053–1058, Dec. 1972.
[20] P. L. Consortium, "The reference p4 program switch.p4," 2017. [Online]. Available: https://github.com/p4lang/switch
[21] M. Reitblatt, N. Foster, J. Rexford *et al.*, "Abstractions for network update," in *ACM SIGCOMM*, 2012, pp. 323–334.
[22] R. Mahajan *et al.*, "On consistent updates in software defined networks," in *12th ACM HotNets*, 2013, pp. 20:1–20:7.
[23] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," in *ACM SIGCOMM*, 2013, pp. 99–110.
[24] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *ACM SIGCOMM*, 2016, pp. 511–524.
[25] D. Saha *et al.*, "Theoretical framework for eliminating redundancy in workflows," in *IEEE SCC*. IEEE, 2009, pp. 41–48.
[26] L. Jose, L. Yan, G. Varghese, and N. McKeown, "Compiling packet programs to reconfigurable switches," in *12th USENIX NSDI*, 2015, pp. 103–115.
[27] M. R. Garey and D. S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[28] B. J. Jain and K. Obermayer, "Extending bron kerbosch for solving the maximum weight clique problem," *CoRR*, vol. abs/1101.1266, 2011. [Online]. Available: http://arxiv.org/abs/1101.1266
[29] E. Aarts and J. Korst, *Simulated Annealing and Boltzmann Machines: A Stochastic Approach to Combinatorial Optimization and Neural Computing*. John Wiley & Sons, Inc., 1989.
[30] D. V. Andrade, M. G. C. Resende, and R. F. Werneck, "Fast local search for the maximum independent set problem," *Journal of Heuristics*, vol. 18, no. 4, pp. 525–547, Aug 2012.
[31] B. Nogueira, R. G. S. Pinheiro, and A. Subramanian, "A hybrid iterated local search heuristic for the maximum weight independent set problem," *Optimization Letters*, vol. 12, no. 3, May 2018.
[32] Xilinx, "Ternary content addressable memory (tcam) search ip for sdnet smartcore ip product guide," 2017.
[33] "The p4visor compiler for bmv2 target," https://github.com/Brown-NSG/P4Visor, 2018.
[34] "P4 software switch (behavioral model) p4-bmv2," 2017. [Online]. Available: https://github.com/p4lang/behavioral-model
[35] "Xilinx sdnet," 2014. [Online]. Available: http://www.xilinx.com/products/design-tools/software-zone/sdnet.html
[36] C. Hu, J. Yang, H. Zhao, and J. Lu, "Design of all programable innovation platform for software defined networking," in *USENIX Open Networking Summit*, 2014.
[37] "The sample p4 programs," 2017. [Online]. Available: https://github.com/p4lang/p4c-bm/tree/master/tests/p4_programs
[38] V. Sivaraman, S. Narayana, O. Rottenstreich, S. Muthukrishnan, and J. Rexford, "Heavy-hitter detection entirely in the data plane," in *3rd ACM SOSR*, 2017, pp. 164–176.
[39] S. Sakai, M. Togasaki, and K. Yamazaki, "A note on greedy algorithms for the maximum weighted independent set problem," *Discrete Applied Mathematics*, vol. 126, no. 2, pp. 313 – 322, 2003.
[40] "The p4 router programs," https://github.com/Brown-NSG/P4Visor/tree/master/FPGAtarget/p4program, 2018.
[41] H. Zeng, P. Kazemian, G. Varghese *et al.*, "Automatic test packet generation," in *8th ACM CoNEXT*, 2012, pp. 241–252.
[42] E. Keller and E. Green, "Virtualizing the data plane through source code merging," in *ACM PRESTO*, 2008, pp. 9–14.
[43] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *9th USENIX OSDI*, 2010, pp. 365–378.
[44] A. Al-Shabibi, M. De Leenheer, M. Gerola, A. Koshibe, G. Parulkar, E. Salvadori, and B. Snow, "Openvirtex: Make your virtual sdns programmable," in *3rd ACM HotSDN*, 2014, pp. 25–30.
[45] X. Jin, J. Gossels, J. Rexford, and D. Walker, "Covisor: A compositional hypervisor for software-defined networks," in *12th USENIX NSDI*, 2015, pp. 87–101.
[46] N. P. Katta, J. Rexford, and D. Walker, "Incremental consistent updates," in *2nd ACM HotSDN*, 2013, pp. 49–54.
[47] R. Mahajan and R. Wattenhofer, "On consistent updates in software defined networks," in *12th ACM HotNets*, 2013, pp. 20:1–20:7.
[48] T. Mizrahi, E. Saat, and Y. Moses, "Timed consistent network updates," in *1st ACM SOSR*, 2015, pp. 21:1–21:14.
[49] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin, "A network-state management service," in *ACM SIGCOMM*, 2014, pp. 563–574.
[50] "In-band Network Telemetry (INT)," http://p4.org/wp-content/uploads/fixed/INT/INT-current-spec.pdf, 2016.
[51] H. T. Dang, D. Sciascia, M. Canini *et al.*, "Netpaxos: Consensus at network speed," in *1st ACM SOSR*, 2015, pp. 5:1–5:7.
[52] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "Kv-direct: High-performance in-memory key-value store with programmable nic," in *26th ACM SOSP*, 2017, pp. 137–152.
[53] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *ACM SIGCOMM*, 2017, pp. 15–28.
[54] S. Luo, H. Yu, and L. Vanbever, "Swing state: Consistent updates for stateful and programmable data planes," in *3rd ACM SOSR*, 2017, pp. 115–121.
[55] A. Sivaraman, A. Cheung, M. Budiu, C. Kim, M. Alizadeh, H. Balakrishnan, G. Varghese, N. McKeown, and S. Licking, "Packet transactions: High-level programming for line-rate switches," in *ACM SIGCOMM*, 2016, pp. 15–28.
[56] G. Gibb, G. Varghese *et al.*, "Design principles for packet parsers," in *9th ACM/IEEE ANCS*, 2013, pp. 13–24.
[57] H. Wang, R. Soulé, H. T. Dang, K. S. Lee, V. Shrivastav, N. Foster, and H. Weatherspoon, "P4fpga: A rapid prototyping framework for p4," in *3rd ACM SOSR*, 2017, pp. 122–135.
[58] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "Virtp4: An architecture for p4 virtualization," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2019, pp. 75–78.
[59] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "Hard virtualization of p4-based switches with virtp4," in *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, ser. SIGCOMM Posters and Demos '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 80–81. [Online]. Available: https://doi.org/10.1145/3342280.3342314
[60] M. Saquetti, G. Bueno, W. Cordeiro, and J. R. Azambuja, "P4vbox: Enabling p4-based switch virtualization," *IEEE Communications Letters*, vol. 24, no. 1, pp. 146–149, Jan 2020.
[61] P. Zheng, T. Benson, and C. Hu, "P4visor: lightweight virtualization and composition primitives for building and testing modular programs," in *14th ACM CoNEXT*, 2018, pp. 98–111.
[62] ——, "Shadowp4: Building and testing modular programs," in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, ser. SIGCOMM '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 150–152.
[63] C. Monsanto, J. Reich, N. Foster *et al.*, "Composing software defined networks," in *10th USENIX NSDI*, 2013, pp. 1–13.
[64] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: A network programming language," in *16th ACM ICFP*, 2011, pp. 279–291.
[65] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett, "Sdx: A software defined internet exchange," in *ACM SIGCOMM*, 2014, pp. 551–562.
[66] M. Canini, D. De Cicco, P. Kuznetsov, D. Levin, S. Schmid, S. Vissicchio *et al.*, "Stn: A robust and distributed sdn control plane," *Open Networking Summit*, vol. 490, 2014.
[67] H. Soni, T. Turletti, and W. Dabbous, "P4Bricks: Enabling multiprocessing using Linker-based network data plane architecture," Feb. 2018, working paper. [Online]. Available: https://hal.inria.fr/hal-01632431
[68] G. P. Katsikas, M. Enguehard, M. Kuźniar, G. Q. Maguire Jr, and D. Kostić, "Snf: synthesizing high performance nfv service chains," *PeerJ Computer Science*, vol. 2, p. e98, Nov. 2016.
[69] H. Chen and T. Benson, "Switch-visor: Towards infrastructure-level virtualization of sdn switches," in *2nd ACM CAN*, 2017, pp. 25–30.
[70] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zupdate: Updating data center networks with zero loss," in *ACM SIGCOMM*, 2013, pp. 411–422.

APPENDIX

A. Complexity Analysis and Proof of the merging problem

Our TDG merging problem can be reduced to and from the Maximum Weighted Independent Set (MWIS) problem: a problem which has been proven to be NP-Complete [27]. Here we provide a sketch of how to reduce our problem to and from the MWIS problem.

We define a function $v(m, i)$ which returns the table from TDG G_m whose table ID is i, thus, $v(m, i) = t_{m_i}$ and $t_{m_i}.tid = i$. To do this reduction, we define a merge candidate set, T_p, as the set of all tables in G_r and G_s that satisfy the equivalence requirement defined in constraint (i).

By definition of constraints (i) to (iii) in Sec. IV-B, all the tables in production and testing programs follow the Lemma 1 (proved in the appendix B).

Lemma 1. *$\forall t_i, t_j \in T_{ms}$, v(s,j) and v(s,i) have*

$$D_r[v(r,i), v(r,j)] \cdot D_s[v(s,j), v(s,i)] = 0 \tag{4}$$

Next, let us construct a new undirected graph $G_p = (T_p, E_p)$ where the vertex set of the graph is T_p and the edge set of the

graph is E_p. Given this definition, we define $\forall t_i, t_j \in T_p$,

$$E_p[t_i, t_j] = \begin{cases} 1 & D_r[v(r,i), v(r,j)] \cdot D_s[v(s,j), v(s,i)] = 1 \\ & or\ D_r[v(r,j), v(r,i)] \cdot D_s[v(s,i), v(s,j)] = 1 \\ 0 & Otherwise \end{cases} \tag{5}$$

Taken together, formulas (4) and (5) provide us with a way to formally reason about the relationship between T_{ms} and G_p. Lemma 2 (proved in the appendix C) provides this relationship.

Lemma 2. *The set of resource-sharing tables T_{ms} is a subset of vertices in graph G_p, no two of which are adjacent, that is, $\forall t_i, t_j \in T_{ms}$,*

$$E_p[t_i, t_j] = 0 \tag{6}$$

Reducing PDP-Merge to MWIS: Lemma 2 restated shows that analyzing graph G_p to identify the set of tables T_{ms} can be reduced to the independent set problem in polynomial time of $O(|T_p|^2)$. Essentially, in constructing G_p, we only keep the dependencies in both D_r and D_s that provide the forward and reverse direction between two nodes. Take nodes *A, D* in Figure 3 as an example, there is a dependency from node A to D in one program as well as a dependency path from D to A in another program. We keep these types of forward and reverse dependencies when creating G_p and delete all others dependencies.

To satisfy our objective of maximizing the shared table resources, we need to find the maximum weighted independent set in graph G_p, known as MWIS problem, an NP-Complete problem [27].

Reducing MWIS to PDP-Merge: Next, we show how to reduce a given MWIS problem to our merging problem. The key lies in transforming a given weighted undirected graph $G_p = (T_p, E_p)$ in the MWIS problem to two weighted DAGs, G_r and G_s, to be merged with the objective of maximizing the weights of the final DAG. More specifically, we can construct the dependencies matrix of two DAGs from G_p as follows:

$$D_r[i,j] = \begin{cases} E_p[i,j] & if\ i > j \\ 0 & Otherwise \end{cases} \tag{7}$$

$$D_s[i,j] = \begin{cases} E_p[i,j] & if\ i < j \\ 0 & Otherwise \end{cases} \tag{8}$$

where $i, j = 0, 1, 2, ..., |T_p|$ are the indices of the nodes in graph G_p. We set T_{ms} as a feasible independent set of G_p. Similarly, with lemma 2, we know that T_{ms} is a feasible set of *resource-sharing* tables when merging two constructed DAGs G_r and G_s. Further, as each node has a weight, solving the maximum weighted independent set of G_p is equal to the identification of the set of tables with maximum shared resource when merging G_r and G_s.

Thus, the maximum weighted independent set (MWIS) problem, an NP-Complete problem [27], can be reduced to our problem in polynomial time $O(|T_p|^2)$. That is to say, merging two weighted DAGs into one weighted DAG with the objective of maximizing weights is an NP-Complete problem.

B. Proof of Lemma 1

We can proof lemma 1 by contradiction. Let us assume that $\exists\ t_i, t_j \in T_m$ so that

$$D_r[v(t,i), v(t,j)] \cdot D_s[v(s,j), v(s,i)] = 1$$

then we know $D_r[v(t,i), v(t,j)] = 1$ and $D_s[v(s,j), v(s,i)] = 1$. As i, j are the ids of the merged tables satisfying the table dependency consistency, according to $Rule1$ we have

$$D_m[v(m,i), v(m,j)] = D_r[v(t,i), v(t,j)] = 1$$

$$D_m[v(m,j), v(m,i)] = D_s[v(s,j), v(s,i)] = 1$$

which means the merging of tables $v(t,i), v(s,i)$ and the merging of $v(t,j), v(s,j)$ introduce a dependency loop to the merged graph D_m. By $Rule2$, D_m is loop free. This is a contradiction. □

C. Proof of Lemma 2

We can proof lemma 2 by contradiction similar with the proof of lemma 1. Assume that $\exists\ t_i, t_j \in T_m$ so that $E_p[t_i, t_j] = 1$, then according to equation (5) we can get $D_r[v(t,i), v(t,j)] = 1$ and $D_s[v(s,j), v(s,i)] = 1$. This will lead to the same contradiction shown in the proof of lemma 1. Hence, we have $\forall t_i, t_j \in T_m,\ E_p[t_i, t_j] = 0$. □

Peng Zheng received the B.S. degree in information security from the Northwestern Polytechnical University, China in 2015. He is currently a visiting researcher at University of Minnesota - Twin Cities, and is pursuing the Ph.D. degree with the Department of Computer Science and Technology, Xi'an Jiaotong University. He has been a visiting research fellow at Duke University and Brown University, in 2017 and 2018, respectively. He has authored papers in CoNEXT, APNet, ICDCS etc. His research interests are computer networking and system.

Theophilus A. Benson received the B.S. degree from Tufts University in 2004, the M.S. and Ph. D. degree in Computer Science from University of Wisconsin, USA, in 2008 and 2012, respectively. He is currently an Assistant Professor in the Computer Science Department at Brown University. His research interests include solving practical networking and systems problems, with a focus on Software Defined Networking, data centers, clouds, and configuration management. In the past, He conducted large scale measurement studies of data centers and enterprise networks; and developed several networked and distributed systems - one of which was purchased in 2012.

Chengchen Hu received the B.S. degree from the Department of Automation, Northwestern Polytechnical University, China, and the Ph. D. degree from the Department of Computer Science and Technology, Tsinghua University, China, in 2003 and 2008, respectively. He worked as an assistant research professor with Tsinghua University from July 2008 to December 2010. After that, he joined the Department of Computer Science and Technology, Xi'an Jiaotong University, China, where he is a full professor. His main research interests include computer networking systems and network measurement and monitoring.