UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Code Merging for Programmable Data Plane Virtualization

Duarte Fonseca Ribeiro Sequeira

Mestrado em Engenharia Informática

Especialização em Arquitetura, Sistemas e Redes de Computadores

Dissertação orientada por:

Doutor Salvatore Signorello

E co-orientada pelo Prof. Doutor Fernando Ramos (IST)

2020

Acknowledgments

After five long years of college, there are plenty of people to thank. Surely I will forget some, but I will try my best.

Firstly, I would like to thank my family. My brother, for setting the example that I have followed my whole life. And my parents, for being the kind, knowledgeable and patient people that you are, and for never letting me throw in the towel.

Secondly, my friends, who continued to deal with me, some for over a decade, even when life put us in different paths. In particular, Pedro, Nuno and Cristina, who have never thought twice before having my back. That next car trip needs to come soon.

Then, to my colleagues, who have become friends I will stick to for the rest of my life. Specially, to Paulo, for being the glue of the group. You don't know how proud I am of you. To João Gil, for being our designated driver, and the only one from "LS", who can relate with me. It's been a pleasure, my friend. To João Lobo, for the meaningful debates. Hope we continue to agree to disagree for many years to come. To Miguel, for the nights we spent working on projects, and for the ones where we didn't work at all. Thanks for the milk, friend.

Finally, a huge thank you to my advisors, Dr. Salvatore Signorello and Professor Fernando Ramos, for guiding me in this adventure and always keeping me on track. It would not have been possible without your constant and insightful help.

This work was supported by FCT through project UPVN: User-centric Programmable Virtual Networks, ref. PTDC/CCI-INF/30340/2017, the LASIGE Research Unit, ref. UIDB/00408/2020 and ref. UIDP/00408/2020, and INESC-ID, ref. UIDB/50021/2020 and ref. UIDP/50021/2020.

Resumo

O desenvolvimento das Redes Definidas por Software (em inglês, Software-Defined Networks, ou SDNs) foi impulsionado pela necessidade de os administradores de redes aumentarem a sua capacidade de controlo. Para tal, as SDNs introduzem a ideia de centralizar o plano de controlo, removendo-o do equipamento de encaminhamento e promovendo a sua desagregação do plano de dados, permitindo agilizar a implementação das políticas de rede.

Apesar de as SDNs permitirem assim programar o plano de controlo, o plano de dados mantém-se inflexível, sendo a sua funcionalidade definida pelos fabricantes de equipamento de rede e mantendo-se inalterável após o fabrico. A implementação de certas políticas leva muitas vezes os administradores de rede a ter necessidade de realizar operações especificas sobre o tráfego da rede, ou mesmo a criar novos protocolos não suportados pelo equipamento. Uma solução para este problema impleica a utilização de *middleboxes*, isto é, hardware específico que é inserido na rede para executar computações que os switches e routers tradicionais não têm capacidade para executar. Este hardware traz, no entanto, algumas desvantagens, nomeadamente o custo de aquisição e o facto de ser especializado para uma única tarefa, forçando os administradores de rede a adquirir diferentes middleboxes para diferentes operações, e a adquirir novas versões caso a funcionalidade desejada se altere. Este processo é dispendioso, lento, e torna a operação da rede ainda mais difícil.

Recentemente, desenvolvimentos ao nível dos chips presentes nos switches, até então apenas capazes de processar pacotes de acordo com o definido pelo fabricante no momento de produção do hardware, permitiram que o processamento de pacotes pudesse ser *programado* pelo utilizador, desta forma introduzindo o conceito de Plano de Dados Programável (PDP). Com a utilização, por exemplo, da linguagem de programação P4, os operadores de rede têm agora a capacidade de desenvolver novos protocolos para o plano de dados, especificando o modo como os pacotes são processados sem recorrer ao plano de controlo e sem a necessidade de hardware especializado.

Existem, no entanto, algumas restrições associadas aos PDPs atuais. Uma destas restrições é o facto de cada switch programável só ter capacidade de correr um único programa P4 de cada vez. Isto traz problemas de modularidade e de eficiência. Por um lado, para ter múltiplas funcionalidades a serem executadas simultaneamente, os administradores de rede são obrigados a produzir programas monolíticos que ficam progressivamente maiores e mais complexos à medida que aumentamos os requisitos. Por outro lado, não é possível integrar múltiplos programas diferentes, potencialmente desenvolvidos por utilizadores diversos, impedindo assim a utilização partilhada dos recursos de hardware e limitando a sua utilização efetiva.

Para colmatar este problema, alguns autores têm vindo a propor soluções de virtualização de PDPs, que têm como objetivo principal permitir aos utilizadores programar diferentes funcionalidades de forma modular e juntá-las de forma dinâmica de modo a correrem simultaneamente no switch. Estas soluções dividem-se em dois tipos: Emulação e Fusão de Código (Code Merging). No caso da *Emulação*, é gerado um programa que utiliza uma série de tabelas (Match-Action Tables) para emular as primitivas básicas do P4. Os programas que contêm as funcionalidades que os administradores desejam implementar na rede são depois traduzidos como entradas nestas tabelas, emulando assim a sua execução como se corressem isoladamente no dispositivo. Esta técnica de virtualização permite que novos programas sejam adicionados e que antigos sejam retirados sem que o dispositivo precise de reiniciar, permitindo a execução de múltiplos programas simultaneamente. No entanto, a emulação é muito ineficiente, requerendo a utilização de inúmeros recursos. Este elevado custo torna esta abordagem impraticável. As soluções de *Fusão de Código* introduzem uma abordagem diferente, que consiste na capacidade de juntar múltiplos programas P4, escritos individualmente, e combinar as suas funcionalidades num só programa, que é depois instalado no switch. O isolamento entre programas, ou seja, a garantia de que um programa não vai interferir com outro de forma não planeada (e.g., acessos concorrentes à mesma zona de memória), é garantido normalmente através da utilização de *tags*, marcadores que distinguem os recursos de cada programa.

As soluções de Fusão de Código têm a desvantagem de gerar um programa que precisa de ser instalado no switch após a sua compilação, obrigando à remoção do antigo programa e a colocação do novo, resultando numa disrupção momentânea devido à paragem do processamento de pacotes. No entanto, esta abordagem apresenta bons níveis de eficiência e reduzido overhead, tornando-a na solução mais efetiva na prática. A solução que representa o estado da arte, o P4Visor, apresenta, no entanto, limitações adicionais. Em primeiro lugar, só permite fundir dois programas muito semelhantes, impedindo dessa forma o desenvolvimento modular e flexível de novas funcionalidades. Adicionalmente, como os switches programáveis possuem restrições relativamente à quantidade de recursos que podem ser utilizados por um determinado programa, é um desafio a integração de várias funcionalidades no mesmo switch. Para mitigar este problema, o P4Visor desenvolve mecanismos que reduzem a quantidade de tabelas utilizadas, mas trata de forma ineficiente do problema fusão dos grafos de parsing dos pacotes, limitando as possibilidades de fusão de código.

Neste trabalho, reconhecendo os problemas de eficiência e desempenho das soluções baseadas em emulação, propomos uma nova plataforma que permita a modularidade de desenvolvimento de múltiplos programas P4 e a sua execução em simultâneo em switches programáveis, usando uma abordagem centrada na Fusão de Código. Esta solução inova relativamente ao estado da arte, ao permitir a integração de múltiplos programas P4 (isto é, mais de dois) desenvolvidos de forma modular, num único programa, e partilhando recursos do grafo de parsing, reduzindo dessa forma a quantidade total de recursos utilizados, atingindo em certos casos melhorias na ordem dos 60%.

Palavras-chave: Redes Definidas por Software, Planos de Dados Programáveis, Virtualização de Redes, a Linguagem P4

Abstract

Recent advances in the hardware capabilities of switching chips have enabled programmability of the data plane. The development of new network protocols and functions, which historically demanded long ASIC design lifecycles to be operated, can now happen quickly and flexibly on Programmable Data Planes (PDPs). Network administrators can directly deploy custom packet processing logic as programs (written in high-level languages such as P4) into their programmable switching ASICs. There are, however, some unresolved problems associated with current PDPs, which hinder their adoption in production networks. One such problem is that a PDP target (e.g., a switch) is currently only capable of running one program at any given time. This limitation has several important consequences. First, it constrains network administrators to write large and complex programs whenever they need to deploy multiple functionalities, which is the common case. Second, it precludes resource sharing between multiple programs, potentially written by different users, limiting resource utilization and thus impacting efficiency.

Inspired by the success story of virtualization in the domain of operating systems, researchers have started proposing solutions to overcome the above issue by virtualizing PDPs. Unfortunately, existing solutions are either very inefficient or lack generality. Hence, this work aims at designing a programmable data plane virtualization platform that enables the deployment of many independently-developed P4 programs on a PDP while introducing the minimum resource overhead. We achieve virtualization at the compiler-level by merging network functionalities into a single, monolithic program, where the individual P4 programs coexist fully isolated from each other. We leverage a state-of-the-art system for code merging, P4Visor, but we improve it by extending the number and variety of P4 programs that can be virtualized, from only two to multiple, and by improving resource efficiency, specifically in the packet parsing module.

Keywords: Software-Defined Networks, Programmable Data Planes, Network Virtualization, the P4 Language

Contents

List of Figures

List of Tables

List of Acronyms

AMD Advanced Micro Devices.

ARP Address Resolution Protocol.

CFG Control Flow Graph.

COTS Commercial Off-The-Shelf.

CPU Central Processing Unit.

DAG Directed Acyclic Graph.

DH Description Header.

HLIR High-Level Intermediate Representation.

IPv4 Internet Protocol version 4.

IPv6 Internet Protocol version 6.

JSON JavaScript Object Notation.

MAT Match-Action Table.

OSG Operation Schedule Graph.

PDP Programmable Data Planes.

PISA Protocol-Independent Switch Architecture.

PVA P4Visor controller Application.

PVC P4Visor Compiler.

PVI P4Visor Interface.

PVM P4Visor Management agent.

RTP Real-time Transport Protocol.

SDN Software-Defined Network.

TCP Transmission Control Protocol.

UDP User Datagram Protocol.

uPVN User-Centric Programmable Virtual Networks.

VLAN Virtual Local Area Network.

Chapter 1

Introduction

The rise of Software-Defined Networks (SDNs) [1] over the past decade has changed the way networks are managed. SDNs feature a logically centralized entity that maintains a logical view of the network and has the ability to reconfigure the network's behaviour by informing switches about new packet forwarding rules. In this way, it becomes simpler for administrators to implement and enforce network policies that had, before SDNs, to be performed using low-level, *ad-hoc* tools that often required per-device human-assisted operations.

Recently, programmable data planes (PDPs) have emerged in the market [2] as a result of hardware developments responding to operator demands, allowing the execution of customized packet processing code on the data plane (e.g., switches) while maintaining comparable performance with legacy fixed-function switches [3]. Network operators and users can now define, in a top-down fashion, the packet processing capabilities of their switching equipment by means of a high-level programming language as P4 [4]. With these tools, operators are now able to deploy, for example, complex congestion-aware load-balancing techniques [5], that analyze the traffic traversing the switch and dynamically change forwarding rules when needed to change path.

The main focus of this work is to investigate techniques to enable multi-tenancy on PDPs; a requirement which is today hardly achievable, but of foremost importance for this programmable networking infrastructure to serve better in production networks. This work is integrated within the context of the User-centric Programmable Virtual Networks (uPVN) project. uPVN aims to confer on users, through a specific platform, the ability to define the packet processing and forwarding in the elements of their virtual networks.

This chapter serves as an introduction to this thesis. The motivation for this work is described in Sec. 1.1. Then, the contributions are summarized in Sec. 1.2. Finally, the structure of the rest of the document is outlined in Sec. 1.3.

1.1 Motivation

Despite bringing important advantages, it is widely accepted that PDPs [2] still have some important problems that limit their full potential. One of such limitations is that they can only run one program (written, for example, in P4 [4], a Domain Specific Language for network data planes) at any given time. Therefore, the common requirement of supporting multiple networking contexts forces developers to write programs that increase both in size and in complexity as more functionalities are introduced. This inherently restricts the flexible and dynamic deployment and composition of network functions.

To mitigate these problems, some researchers have proposed virtualization solutions [6, 7, 8, 9], breaking the hardware's single-program restriction and allowing for multiple programs to be deployed simultaneously. These virtualization solutions can be broadly classified into two different categories, namely Emulation-based and Code Merging. The former class of solutions consists in generating an *uber* P4 program that emulates the device's hardware, serving as a platform for the deployment of multiple other programs. To do so, the emulator allocates resources on the target architecture (e.g., match-action tables on the switch's pipelines) for each one of P4's primitive actions. The use of primitives by other programs can then be translated into entries to these tables in the control plane. This way, emulator-based solutions allow for multiple programs to run simultaneously by populating the emulator's tables with the translated entries. Through such a mechanism, the emulator enables the addition and removal of programs in run-time (*seamless reconfiguration*). Yet, despite this advantage, the emulator's resource usage overhead introduces a very large performance penalty, making the deployment of such solution impractical.

The second type of solutions takes multiple P4 programs as input, merges and compiles them into one monolithic program that is finally deployed on the PDP. This approach leverages the fact that it is possible to share resources between different P4 programs when those have equivalent program functionalities, which is a common case. Code merging solutions orchestrate through the introduction of a small amount of resources the order in which the different input programs must execute within the execution flow of the merged program. To maintain isolation between the programs (i.e., input programs cannot access resources in the merged program that they do not originally instantiate), the resources of each individual programs are tagged, to solve ambiguities (e.g., resources with the same name belonging to different programs). Once the resources have been correctly tagged, they are merged into a single, larger program, alongside with the aforementioned virtualization-specific resources. However, the state of the art in Code Merging, P4Visor [8], restricts the number of programs that can be merged to only two, and to be very similar programs. Thus, it fails to provide a platform that truly allows developers to deploy various networking functionalities without the need to write large and complex P4 programs. Furthermore, as the number of functionalities deployed increases, so does the amount of resources used. For this reason, an efficient resource sharing mechanism is fundamental, minimizing the resources used while guaranteeing *isolation* and *correctness* (i.e, the merged program does not differ in functionality from the input programs). P4Visor also presents limitations on that respect, namely by employing an inefficient mechanism for packet parsing.

1.2 Contributions

With our work, named P4Visor++, we improve the state of the art in PDP virtualization by tackling limitations in the existing Code Merging solutions to avoid some performance and inflexibility issues associated with those systems.

More precisely, we have improved upon the system in [8], P4Visor, by allowing the merge of multiple programs (i.e., more than two) and by designing a new parse graph merging technique. Our design was driven by the empirical observation that the mechanism used by P4Visor to merge the parse graphs from the different programs does not optimize the resources required to store the merged graph. As a consequence, merged graphs produced by [8] cannot be practically complied and executed on certain programmable targets, because of their large sizes, for relevant common use-cases.

In summary, the main contributions of this work are the following:

- the design and implementation of an algorithm that merges parse graphs from several input P4 programs by sharing equivalent program blocks.
- the design of mechanisms to enforce correctness and isolation properties for the input programs in the merged parse graph.
- an extensive evaluation of our merge algorithm performed using several real P4 programs, through a test suite developed for this work.

1.3 Structure of the document

The rest of this thesis is organised as follows:

- Chapter 2 - Related Work
  - This chapter first introduces some fundamental concepts about the P4 language for packet processors and the target programmable switching architecture, then it reviews the state of the art on PDP virtualization.
- Chapter 3 - Design
  - This chapter details our algorithm for merging parse graphs in P4 programs, illustrating the design rationale. It presents the criteria defined to establish equivalence between headers and between parse states, the algorithm designed to correctly merge those equivalent program's elements and the mechanisms used to optimize resource usage in such a process.
- Chapter 4 - Implementation
  - This chapter describes the implementation of our solution built by extending the software provided with [8]. Our software implements the mechanisms used to determine:

  (i) how headers and parse states can be shared, (ii) how the original parser states and transitions must be modified in order to preserve *correctness* and *isolation* within the merged graph, and (iii) how the merged graph can be further optimized to save resources.

- Chapter 5 - Evaluation
  - This chapter first illustrates the development environment and the P4 programs used for testing purposes. Then, it presents an extensive evaluation of our work, with the correctness of our merging algorithm being tested and our resource usage results being compared against the state-of-the-art.
- Chapter 6 - Conclusion
  - This chapter summarizes the work done in this thesis, followed by a discussion of future work.

Chapter 2

Related Work

The emergence of programmable data planes has brought a new level of flexibility and control to large-scale network administrators. However, the devices' inability to provide multi-tenancy represents an open issue that some have attempted to mitigate.

Across the state of the art, there exist two main approaches to network data plane virtualization, which we respectively name *Emulation-Based* and *Code Merging*, for reason that will be made clear in this chapter. The ultimate goal of both approaches is to create a platform that enables multiple P4 programs to be executed simultaneously on the same physical PDP. Both approaches also aim to guarantee important properties such as *isolation* between programs, that is, each program should operate without its functional and non-functional properties being altered by the other programs; and *correctness*, that is, the merged program does not differ in functionality from the input programs. In this chapter, we first provide an introduction to the P4 language and to a P4-programmable target's architecture, and, afterward, we illustrate these two virtualization approaches, by presenting the existing related systems and techniques. To conclude this chapter, we present a comparison between the state-of-the-art PDP Virtualization solutions, focusing on the limitations that motivate our work.

2.1 Target Architecture and P4 Language

The switching architecture widely used in most of today's programmable high-speed switches is the Protocol-Independent Switch Architecture (PISA) [10], which is illustrated in Figure 2.1. The functionality of this switching ASIC is not bound to recognize and process a fixed set of standard protocols defined at chip design-time. Rather, it can be entirely defined and reconfigured by a P4 program. A P4 program defines the packet processing logic which is compiled down into, and executed by, the PISA switch's hardware. In order to achieve Terabit speeds, PISA switches process packets using a feed-forward data pipeline, i.e., processed packets follow a path of execution that always moves forward along the next stages of the pipeline, with strict, deterministic timings. The switch's pipeline is composed of Match-Action Tables (MATs) stages programmable through two main control blocks, one block for *Ingress* processing and one block for *Egress* processing. Tables in MAT stages can be populated by control plane logic through a program-independent API, such as P4Runtime [11]. PISA switches can also maintain state across multiple packets by using some stateful memory per-stage, called *registers* in the P4 language. Registers can be accessed and modified along the processing pipeline.

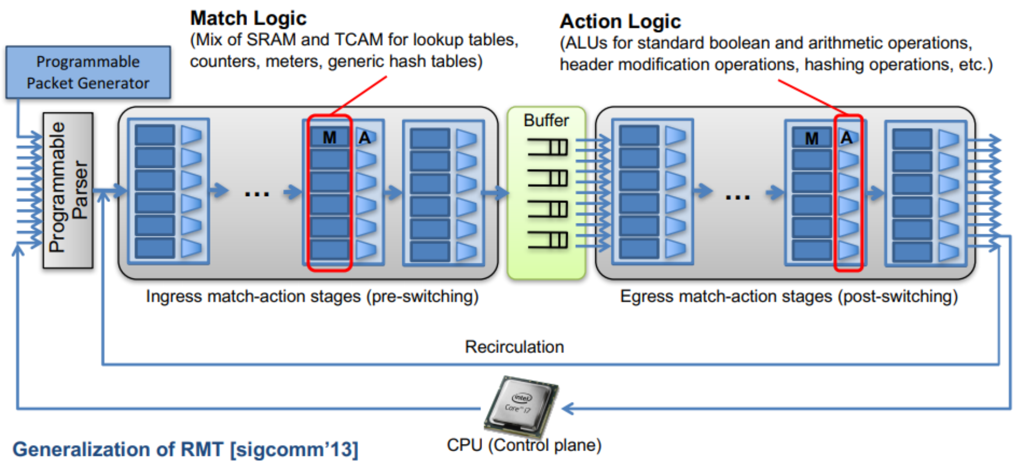
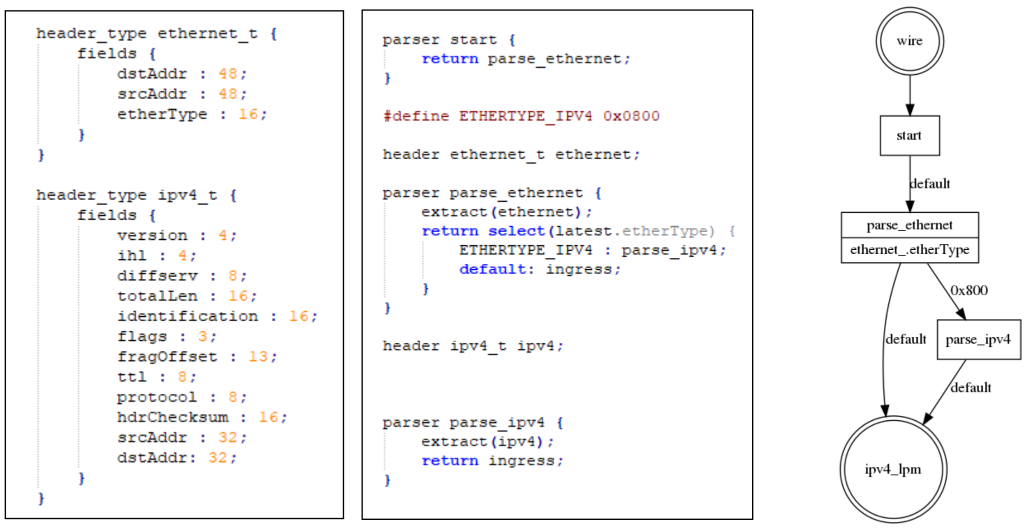

Match Logic
(Mix of SRAM and TCAM for lookup tables, counters, meters, generic hash tables)
Action Logic
(ALUs for standard boolean and arithmetic operations, header modification operations, hashing operations, etc.)
Programmable Packet Generator
Programmable Parser
M
A
Buffer
Ingress match-action stages (pre-switching)
Egress match-action stages (post-switching)
Recirculation
CPU (Control plane)
Generalization of RMT [sigcomm'13]

Figure 2.1: Protocol Independent Switch Architecture (PISA) model

The P4 language [4] allows developers to define the packet processing logic of a PDP through a high-level program. At first, a P4 program defines the format of the packet headers, stating the size and fields of each header, and a sequence of operations to extract the defined headers and their fields from the received packets for further processing. This P4 Parser block specifies the order in which the protocol headers must be extracted, by instantiating parse states (blocks responsible for extracting headers) and defining conditional transitions between such states (e.g., an IPv4 parse state can only transition to a TCP parse state if the extracted IPv4 header contains a value of 6 in its `protocol` field). An example implementation of a parser can be seen in Figure 2.2 (b), with the related headers definition reported in Figure 2.2 (a). An illustration of the resulting parse graph for this program can also be seen in Figure 2.2 (c).

The main packet processing logic is implemented in P4 through control blocks (an example is provided in Figure 2.3), where MATs and the sequence of their execution are expressed. A MAT consists of two kinds of functional parts: one table and one or more actions as seen in Figure 2.3. A match table is defined by matching fields (which can be protocol fields or packet metadata), match types (e.g., exact, ternary, longest-prefix match) for its matching fields and associated actions. Actions are fragments of code performing modifications on the packet data, metadata and stateful memory, whose executions can be triggered by packets matching tables entries. The control plane logic, running in the switch CPU or in an external controller, is responsible for populating the MATs defined in the P4 program with table entries, binding matches with actions and optionally with actions data. For example, in Fig. 2.3, if an IPv4 destination address matches to a certain prefix, the table `ipv4_lpm` will execute the `ipv4_forward` action to set the correct forwarding *egress* port for that packet.



```
header_type ethernet_t {
    fields {
        dstAddr : 48;
        srcAddr : 48;
        etherType : 16;
    }
}

header_type ipv4_t {
    fields {
        version : 4;
        ihl : 4;
        diffserv : 8;
        totalLen : 16;
        identification : 16;
        flags : 3;
        fragOffset : 13;
        ttl : 8;
        protocol : 8;
        hdrChecksum : 16;
        srcAddr : 32;
        dstAddr: 32;
    }
}
```

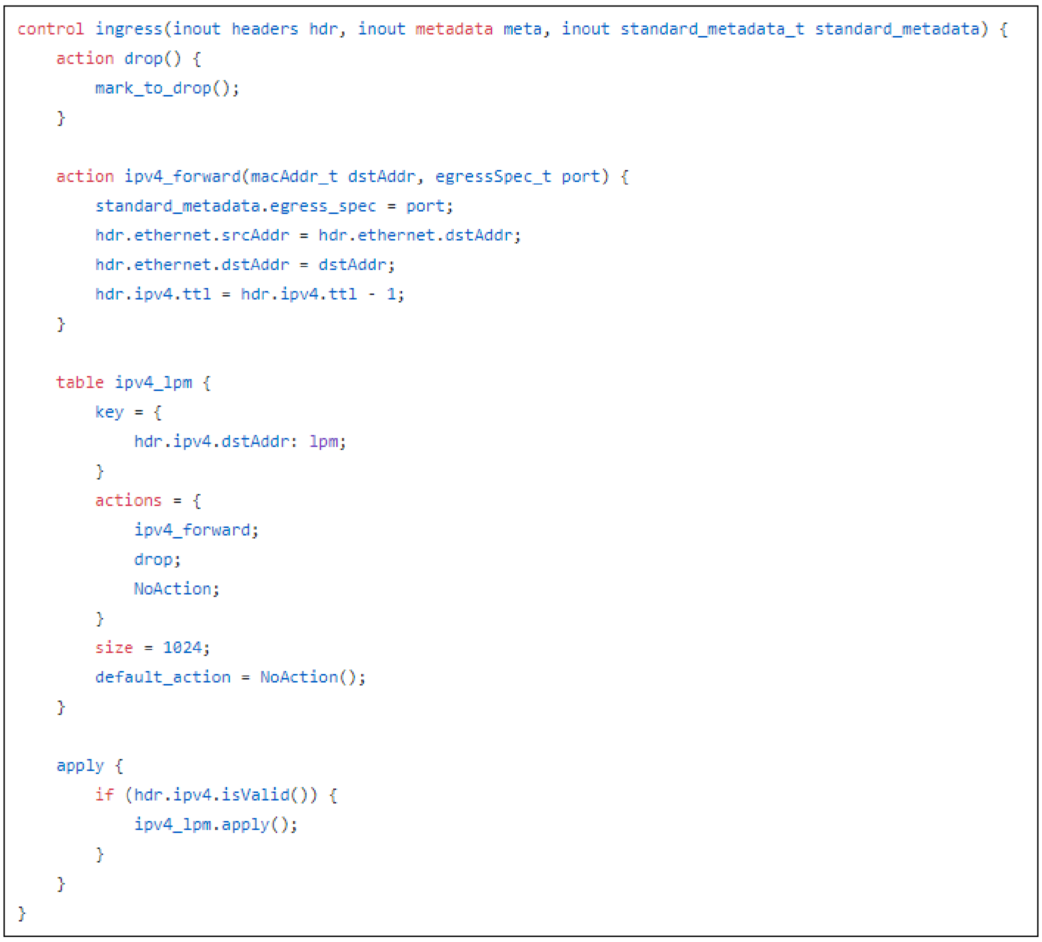
(a)

```
parser start {
    return parse_ethernet;
}

#define ETHERTYPE_IPV4 0x0800

header ethernet_t ethernet;

parser parse_ethernet {
    extract(ethernet);
    return select(latest.etherType) {
        ETHERTYPE_IPV4 : parse_ipv4;
        default: ingress;
    }
}

header ipv4_t ipv4;


parser parse_ipv4 {
    extract(ipv4);
    return ingress;
}
```

(b)

(c)

Figure 2.2: Example of headers (a) and parser (b) definitions in $P4_{14}$, for a simple Ethernet/IP protocols stack. In (c), an illustration of the resulting parse graph.



```
control ingress(inout headers hdr, inout metadata meta, inout standard_metadata_t standard_metadata) {
    action drop() {
        mark_to_drop();
    }

    action ipv4_forward(macAddr_t dstAddr, egressSpec_t port) {
        standard_metadata.egress_spec = port;
        hdr.ethernet.srcAddr = hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr = dstAddr;
        hdr.ipv4.ttl = hdr.ipv4.ttl - 1;
    }

    table ipv4_lpm {
        key = {
            hdr.ipv4.dstAddr: lpm;
        }
        actions = {
            ipv4_forward;
            drop;
            NoAction;
        }
        size = 1024;
        default_action = NoAction();
    }

    apply {
        if (hdr.ipv4.isValid()) {
            ipv4_lpm.apply();
        }
    }
}
```

Figure 2.3: Example of a P4 control block consisting of a single table matched only by valid IPv4 packets.

2.2 Emulation-based approach to PDP Virtualization

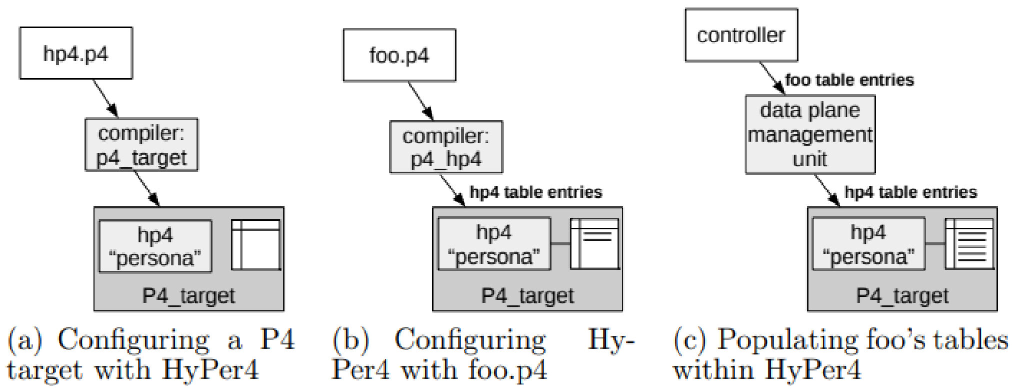


Figure 2.4: Emulation-Based illustration from [6].

Emulation-Based solutions consists of a "uber" P4 program which emulates the device's hardware to execute more than one P4 program simultaneously. The emulator program, which physically sits between the hardware and any deployed program, manages the resource usage for each of the latter, giving it the illusion that it is operating alone and directly on the hardware. These virtualization solutions usually include two main software modules: i) a compiler, responsible for generating the emulator program and for translating P4 programs into table entries for the emulator, and ii) a management unit, responsible for translating programs' run-time configuration to entries of the emulator's tables. An illustration of the workflow of these approaches can be seen in Figure 2.4.

Emulation-Based not only provides a platform for multiple programs running simultaneously, but it also enables an invaluable feature to PDPs: seamless reconfiguration. Normally, changes to the deployed P4 functionality requires rebooting the PDP device so that some new compiled code can be loaded. This represents a problem, as maintaining devices available at all times is highly desirable. Featuring seamless reconfiguration, Emulation-Based allows PDPs to introduce and remove programs in run-time without service disruption, for as long as changes to the main program (the "emulator") are not required.

2.2.1 HyPer4

HyPer4 [6] represents the seminal work in the area of virtualization platforms for P4-programmable PDPs. The system consists of a single *generic* P4 program virtualizing the target PDP and working as an emulator for other P4 programs.

The HyPer4's program declares a number of tables which depends on the complexity of the programs to be emulated. More precisely, the emulator program provides a platform that declares multiple tables for each P4 primitive action. This way, the use of primitives in a native P4 program is now translated into a set of entries that will then populate the emulator's tables. To guarantee isolation between different programs, each of the emulator's tables contains a special ID that

uniquely identifies what original program each table entry belongs to. At the reception of each packet, a metadata field in the packet is associated to one of those IDs via some deterministic process, e.g., based on the ingress port or on a specific protocol field within the packet, to ensure that only one emulated program executes certain table entries.
Additionally, HyPer4 allows for *Virtual Networking*, that is, the chaining of different emulated programs. This is achieved by changing the program's ID in the metadata field and by recirculating the packet (operation achieved leveraging a target's primitive), once that has been processed by the current program. This process achieves sequential processing, allowing for multiple functionalities to be applied to the same packet without the need for multiple physical devices.

Despite some advantages, HyPer4's virtualization platform suffers from some serious drawbacks. The main one is intrinsic to the Emulation-Based approach. Specifically, the emulator program must contain a large number of tables to emulate other programs, leading to a significant overhead when compared to native P4 programs. Furthermore, the periodic use of the recirculation primitive, used by HyPer4 to implement both a generic parser of the emulated programs and the service chaining mechanisms, may considerably reduce the overall throughput, as new packets are prevented from entering the pipeline by the recirculated packets (at full traffic rate, each recirculation step represents halving the throughput, and in normal conditions HyPer4 requires multiple recirculation steps). Finally, despite providing isolation among the deployed programs, HyPer4 does not provide CPU isolation, hence it does not guarantee that every action can be completed within a CPU cycle. As potential consequence, programs can cause stalls in the pipeline and so delay other programs' execution.

Overall, HyPer4 does not provide a practical solution for the PDP virtualization problem, since it introduces a performance loss which is undesirable for most deployments in production networks. Anyway, it represented the first attempt to showcase the benefits of programmable data plane virtualization and fostered other related works.

2.2.2 HyperVDP

Similarly to HyPer4, HyperVDP [7] emulates multiple P4 programs using a single P4 program as operations and resources manager. HyperVDP improves upon the HyPer4 emulation technique with regard to performance and efficiency. HyPerVDP addresses those issues by avoiding part of the recirculation reducing the amount of resources, mainly MATs, used to emulate generic packet processing on the PISA pipeline.

The first issue with HyPer4 tackled by HyperVDP is the use of recirculation for parsing. To extract a sequence of header types, HyPer4 identifies the size and position of a header, extracts it and then resubmits the packet to the ingress so that the process can be repeated for the following headers. This recirculation of packets introduces a significant reduction of the overall device's throughput.To avoid recirculation, HyperVDP encapsulates each packet with a description header (DH) that contains the size of the header and the ID of the program that will process it. In so doing, HyperVDP can extract the entire headers stack and then parse it without recirculating the packet. This process, named “rapid parsing” in HyperVDP, trades an increase in performance for the extra space required by encapsulation.

HyperVDP also optimizes the placement of the emulated programs at the stage level in the pipeline. To achieve this goal, it develops a technique called “Control Flow Sequencing”, a process that creates a DAG (Directed Acyclic Graph) of all the stages of a program as nodes and the path between stages as branches. Graphs from multiple programs can then be mapped into slots, similarly to how virtual memory is mapped to physical one. This mapping is done accordingly not to create inner loops which would alter the correct flow of execution of the executed programs. If, at the end of the mapping process, the number of stages exceeds the number of available slots, HyperVDP uses the recirculation primitive to processes the rest of the graph.

Finally, HyperVDP reduces the number of tables used for such a purpose by performing aggregation into compound actions. In contrast, HyPer4 uses a predefined number of tables for each primitive, requiring more resources to scale up the number of supported primitive actions.

The optimizations in HyperVDP considerably improve performance and efficiency upon HyPer4. The experiments reported with HyperVDP show, in the best case, an increase of 466.3% in throughput, and on average a decrease of 26.5% in delay. However, when compared with running native P4 programs individually, HyperVDP shows an increase between 27 to 41% in delay and a decrease between 34 to 70% in throughput, proving that the performance issues associated with Emulation-Based approaches are still very present. In conclusion, the drawbacks of this kind of virtualization solution make very hard any deployment in practice, especially since other techniques (see Sec. 2.3) offer similar properties with largely better performance and efficiency.

2.3 Code Merging approach to PDP Virtualization

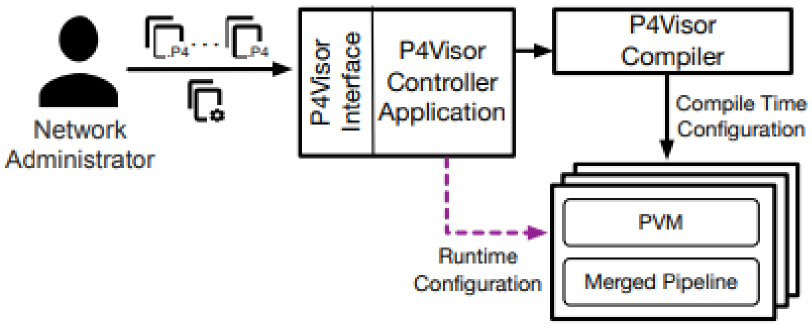


Figure 2.5: Illustration, from [8], of the P4Visor system for the virtualization of P4-PDPs through Code Merging.

Often called *lightweight virtualization*, code merging solutions tackle the problem of running multiple P4 programs on the same PDP target by merging them into one monolithic program at compile-time. These solutions are less flexible than the Emulation-Based ones since they do not support seamless reconfiguration. The reason is that deploying a new program on a PDP device implies replacing the current one, forcing a reboot of the target and a consequent downtime for that change to occur. However, these solutions use significantly less resources and their overall performance considerably improves. As for Emulation-Based approaches, these *lightweight virtualization* platforms usually feature, besides a compiler responsible for merging the programs, a management unit responsible for translating the original programs' table entries into the merged program's table entries.

2.3.1 P4Visor

The P4Visor virtualization platform [8], illustrated in Fig. 2.5, consists of four main components: PVI (P4Visor Interface), PVC (P4Visor Compiler), PVM (P4Visor Management agent) and PVA (P4Visor controller Application). The PVI is the tool used by the developer to manage how the different P4 programs will interact. Given the programs and specifications inserted in PVI, the PVC will generate a monolithic, merged P4 program and an additional "P4VisorConfiguration" file. This file contains the mapping between original and virtual IDs that distinguish the resources used by each program. This is necessary because table IDs are unique for each program, but when the programs are merged, some IDs may be reused. This mapping between old and new IDs allows the administrators to correctly insert the run-time configuration into the PDP, for example, adding an entry to a program's table. This task is performed by the PVM, an agent that runs on the PDP device, and uses the IDs mappings to multiplex and demultiplex messages between the control plane and the PDP.

To merge the parse graphs of two P4 programs (the maximum supported), P4Visor analyses the DAGs of each program and merges them. This is performed by introducing a specific flag, resulting in a shared node proceeding to nodes that exist only in the DAG of one of the programs. The introduced flag defines whether or not a packet should be processed by a certain node in the merged graph. The same logic is applied to share tables and actions between the two programs.

Besides merging only two programs, P4Visor does not deal with target dependency, that is, it does not consider possible constraints placed on the merging process by the target's architecture, such as available number of pipeline stages and memory resources. Besides this issue, P4Visor does not support seamless reconfiguration and so, when a new program needs to be merged, or simply an old one needs to be modified, the merged program must be entirely recreated, compiled and loaded onto the target. As we explained in previous sections, this operation implies a downtime for the target.

2.3.2 P4Bricks

P4Bricks [9] differs from P4Visor an the remaining virtualization approaches by providing a different form of multi-tenancy. More precisely, P4Bricks allows for independently-written P4 programs to process the incoming traffic simultaneously, rather than deciding which functionality must process each packet. P4Bricks consists of two main components, Linker and Runtime, responsible respectively for merging the programs and updating the table entries. A high-level illustration of

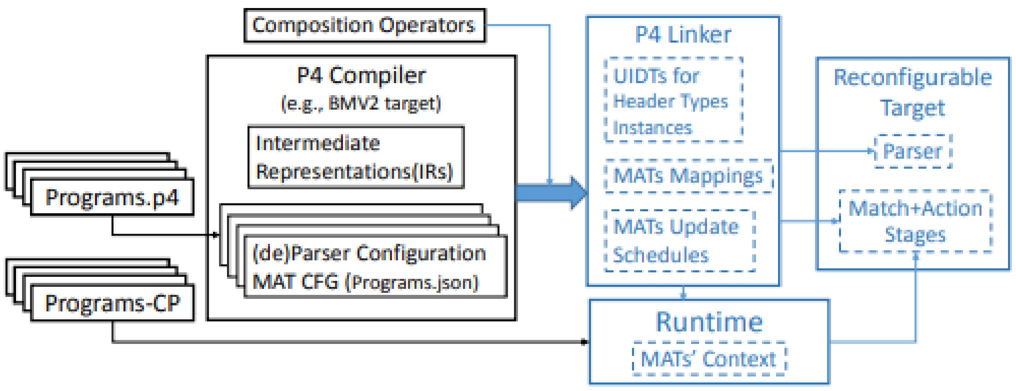


Figure 2.6: Illustration, from [9], of the P4Bricks system for the virtualization of P4-PDPs through Code Merging.

the P4Bricks system is provided in Figure 2.6.

In the Linker, P4Bricks identifies equivalencies between the header types used in each program, with the aim of merging the programs' parsers. Each header type is given a unique ID and then, for each program, the Linker maps the headers to those IDs and stores this mapping into a table. Intrinsic metadata associated with each packet can also be shared across multiple programs as it is related with the target's architecture, but user-defined metadata, as it is specific to each program, is given a different ID using the program's name to guarantee isolation. To merge the parsers, the Linker looks for equivalent parse states between programs. To be considered equivalent, the parser states must satisfy three conditions: i) must extract from the same bit index in the packet, ii) must advance the same number of bits and extract the equivalent header types (i.e., headers that have the same number of fields, with each field having the same width) and iii) if both states have "select" expressions, the lookup fields must be equivalent (the *select* operation allows for conditional branching to different states, and so the fields must be the same to ensure that equivalence is maintained). For example, relatively to point iii), two states from different input programs, both extracting an Ethernet header, will not be equivalent if one of them *selects* on the *etherType* of the header, while the other *selects* on the destination address. Deparsers, unlike parsers, are not represented with DAGs in the P4 language. For this reason, P4Bricks merges deparsers by using the parsers of the programs and creating from those DAGs that can be merged. As a result, P4Bricks can only merge programs that use the same networking protocol stack to parse and deparse. This approach has the drawback that cannot merge correctly programs performing packet encapsulations, since those new layers cannot be inferred by the initial parser.

As explained in Section 2.1, each program has its own pipeline, that is, its sequence of MATs. This sequence is represented by another DAG, named Control Flow Graph (CFG), where each node is a MAT and the edges are transitions between the different MATs. The P4Bricks Linker analyzes, for each match and action of a program, the order of accesses to resources and the operations performed on them (read/write), and proceeds to create an Operation Schedule Graph

(OSG). An OSG is therefore a graph that represents all the accesses made to a certain resource during the execution of the program. Once each program's CFGs and OSGs have been computed, the Linker merges all CFGs into one new CFG and one OSG for each resource, and then maps them to the physical memory capacity in the stages, while respecting the operation orders. Two techniques are used to facilitate the mapping of these graphs: decomposing merged MATs into sub-MATs and allowing out-of-order writes in the OSGs. Sub-MATs are necessary because there may not be enough hardware capacity to execute all the operations present in a stage, and so the Linker divides the operations so that they can occur in the subsequent stage, while maintaining referential integrity between them. Out-of-order writes are implemented to prevent reads to a certain resource from happening after a write that the original OSG does not contain.

Once the merged program is deployed, the Runtime component is responsible for translating the table updates from the control plane of the original programs into entries for the tables of the merged program. Special attention is devoted to sub-MATs , since an update to a MAT must reflect to the related sub-MATs as well.

In theory, P4Bricks allows for two modes of composition, sequential and parallel, that the other solutions cannot provide. To perform sequential composition (that is, a packet processed by a sequence of programs), P4Bricks merges and maps the CFGs and OSGs to the same pipeline, avoiding the performance drop of recirculation. P4Bricks supports parallel composition by imposing hard locks onto the target's resources.

Unlike P4Visor, P4Bricks [9] is not available open-source and does not provide experimental results[1]. Anyway, it provides valuable insights to address this problem.

2.4 Analysis of the state-of-the-art PDP Virtualization Solutions

To better illustrate the main differences between the state-of-the-art approaches, we present a summary in Table 2.1, alongside a detailed description of each feature presented.

2.4.1 Number of Programs Supported

This metric represents the number of P4 programs that can be deployed simultaneously on a device using each virtualization solution. For the purpose of compiling this table, the data here reported came from the original experiments performed in those works. There is not, in any of these approaches, a theoretical upper bound on the number of programs that can be deployed simultaneously. Yet, when "hardware limited" is indicated, both hardware capabilities and program complexity are important factors defining a practical upper limit.

2.4.2 Seamless Reconfiguration

This category refers to the ability that administrators have to add or remove programs from the data plane without disrupting the "normal" functioning of the PDP target. In the "Emulation-Based"

[1]P4bricks was presented as a technical report and was not peer-reviewed. After contacting the authors we understood the project was put on hold for undetermined time.

	HyPer4	**HyperVDP**	**P4Visor**	**P4Bricks**
Type	Emulation Based	Emulation Based	Code Merging	Code Merging
#Programs Supported	Hardware limited (examples with 8)	Hardware limited (examples with 5)	2	Hardware limited (examples with 2)
Seamless reconfig.	Yes	Yes	No	No
P4 Version	$P4_{14}$	$P4_{14}$	$P4_{14}$	$P4_{16}$
Parallel Processing	No	No	No	Partially
Sequential Processing	Via Recirculation	Via Recirculation	Via Recirculation	Yes
Shared Tables	Yes	Yes	Yes	No
Delay Increase (vs P4)	70 to 90%	27 to 41%	<1% <3%	Not Available
Throughput Drop (vs P4)	80 to 90%	34 to 70%	<1% <1.5%	Not available

Table 2.1: Comparison of the virtualization systems in terms of selected features and offered performance. The values for delay and throughput are taken from [8] and [7].

approaches, programs can be removed or added without causing any downtime to the other running programs, the only exception being the emulator program. The “Code Merging” approaches, on the other hand, require the programs to be merged and compiled before deployment. Therefore, the addition or removal of any program requires modifying the deployed merged program and loading a new one, forcing a reboot of the device.

2.4.3 P4 Version

This category indicates the version of the P4 language supported by each virtualization solution. As $P4_{16}$ is a relatively new version, most approaches were designed and implemented to support the earlier $P4_{14}$ language version. Among the four approaches, only P4Bricks supports $P4_{16}$ programs.

2.4.4 Parallel and Sequential Processing

Parallel processing refers to the ability to have multiple programs processing a packet simultaneously through the same pipeline stages. The Emulation-Based solutions do not allow this, as each emulated program executes in isolation from the others. Differently, with the code merging solutions different operation modes can be specified. In P4Bricks, *parallel processing* is allowed as long as access to shared resources is restricted (if the traffic flows are disjoint, meaning there are no shared resources, no such restriction is required). For example, multiple programs are merged if only one of these programs writes to a certain resource, and every read performed by other programs on this same resource happens before that write. For this reason, we consider that P4Bricks only partially allows this mode of operation.

With regards to sequential processing, all the solutions analyzed by our work provide this feature. Sequential processing refers to the ability to state an order of execution for the programs, which will be applied on a packet basis. The way this is achieved in the Emulation-Based solutions and in P4Visor is by using recirculation. Recirculation is a commonly available device primitive allowing for packets to be processed multiple times across the same processing pipeline. This primitive is very useful to apply a different program at each recirculation, but it decreases the overall throughput. P4Bricks enables this operation mode by merging the operation graphs of different programs, forcing a sequence to be followed by linking the end of a program to the beginning of another. If the hardware supports the number of physical stages required by the merged graph, multiple programs can, in the end, be executed in the same pipeline without recirculation being used.

2.4.5 Shared Tables

Different programs may have equivalent tables, so instead of separately allocating resources for each program, the equivalent tables can be shared. This process eliminates redundancy and optimizes tables usage. With the Emulation-Based solutions, this optimization is achieved by design, as emulated programs are restricted to using the tables instantiated by the emulator. In the Code Merging solutions, tables are shared when an equivalence is established during the merging process. From the two Code Merging techniques presented, only P4Visor performs this optimization.

2.4.6 Delay and Throughput

These two metrics refer to the performance loss introduced by each virtualization solution when compared to running P4 programs natively. The evaluation methodology varied across the analyzed systems. In HyperVDP and HyPer4 the tests consisted in deploying simultaneously different programs (several combinations of different types and number of programs per test) and then comparing the results with the same overall packet processing logic combined into a single native P4 program. For P4Visor, two random programs are picked to be merged and then the performance of a program in the merged context is compared with the performance of the same program written in a single P4 program.
The results show that, with regards to delay, the Emulation-Based approaches are at best 27% and at worst 90% slower, while throughput results show a 34 to 90% drop in relation to native P4 programs. The cost is therefore very high. P4Visor introduces a smaller overhead, ranging from 1.5 to 3% in software switches and an overall 1% drop in hardware. There is no evaluation reported in the P4Bricks work.

2.5 Summary

In this chapter, we provided a brief introduction to the P4 language and to the target architecture used in programmable switches in Section 2.1. Then, in Sections 2.2 and 2.3, we overviewed the two main approaches to data plane virtualization, followed by a summary of the most relevant works in this area. Finally, in Section 2.4 we provided a detailed comparison between the state-of-art PDP virtualization solutions.

Chapter 3

Design

In this chapter, we present the design of our system to run multiple programs on the same P4 programmable data plane. In particular, we illustrate and explain the design choices that we made in order to correctly merge parse graphs from multiple P4 input programs.

Our design introduces a custom header, used to track which portion of the merged parse graph must process packets belonging to each program, thus allowing a potentially unrestricted number of input P4 programs to be deployed concurrently. Furthermore, our design achieves an efficient parse graph merge by facilitating resource sharing, while performing minor modifications to the input programs in order to guarantee *correctness* and *isolation*. Although our work considers the $P4_{14}$ language specification (as that is the language version supported by the state-of-the-art system we targeted to improve upon), all design choices described in this thesis are easily applicable to the newer $P4_{16}$ version of the P4 language.

This chapter is organized as follows. Firstly, in Section 3.1, we enumerate the requirements of our solution. Secondly, in Section 3.2, we dissect some design choices made by the state-of-the-art P4Visor [8] system, to highlight the limitations of that solution that motivate our work. In Section 3.3, we present a high-level description of our system. In Section 3.4, we define P4 header equivalence, a core concept that our algorithm leverages in the merge process. In Section 3.5, we specify the criteria that must be met for two parse states to be shared, followed by a description of a mechanism for state transitions to be correctly merged. Finally, in Section 3.6, we describe how our algorithm detects and removes duplicate transitions from shared states in order to optimize the resources used in the merged program.

3.1 Requirements

In the process of designing our virtualization solution, we first established a series of requirements that must be met by our platform.

- Our solution must allow for an unrestricted[1] number of independently-written P4 programs to be integrated into the final merged program.

[1]By unrestricted we mean that any restrictions that limit the number of merged programs arises not from our design, but from constraints of the underlying hardware (e.g., memory) or the network (e.g., header space).

- Our solution must ensure the programs are functionally equivalent and fully isolated from each other, effectively behaving as if they were deployed standalone in the target switch.
- Our solution must optimize resource usage, as hardware switches have scarce amounts of resources available. The necessity for this optimization becomes more evident as the number of virtualized programs increases.
- Our solution must introduce a minimum amount of platform-specific resources (that is, not present in the input programs) in the merged program. This requirement is specially important for preventing performance losses relatively to natively-deployed P4 programs, as occurring with emulation-based virtualization approaches.

3.2 Limitations of the State of the Art

To be able to fulfill one of our main requirements - efficiency - we have decided not to resort to emulation-based approaches, as they require an excessive amount of resources to operate [6][7]. Our solution thus opts for a code merging approach for virtualization of the switch data plane.

Our work is motivated by some limitations of the state-of-the-art code-merging solution P4Visor [8]. After performing an in-depth analysis of this system, including the published paper [8] and the available code base [12], we identified issues that demonstrate P4Visor does not guarantee a correct and resource-efficient merging of P4 programs in some cases. In addition, P4Visor is only able to merge two programs. In this section, we present in more detail some of those limitations.

3.2.1 Incorrect Merge of Header and Metadata Structures

When merging different P4 programs, the headers and metadata structures present in each individual input program must be carefully merged to ensure their correct and unambiguous representation in the merged program. By analysing P4Visor's code, we can observe that the merge of these program elements is achieved by updating the merged program's intermediate representation (HLIR) with the headers from the input programs. More precisely, P4Visor first generates the HLIR of two input programs (this system only works with two input programs) and only afterward it merges the contents of both objects into a third HLIR object.

In the HLIR object, headers and metadata are stored in a dictionary, with the name of the header used as a key and the corresponding header object stored as the value. By updating the merged HLIR's header dictionary with the input programs, P4Visor adds every header with a unique name to the merged program, and replaces pre-existing headers in case the names are equal. In practical terms, this means that *P4Visor establishes equivalence between headers based on their names in the original programs, which are usually arbitrarily given by developers.* Therefore, if two headers have the same name, they are shared. Otherwise, if their names are different, the headers are not considered equivalent and so they are not shared.

The above design choice raises the following issues. Firstly, headers and metadata that have different structure organizations (i.e., different fields and different field widths) will be shared across programs if they have the same name. More precisely, the last header definition to be merged will be the one included into the final HLIR object (as each update replaces any existing header with the same name). For example, an IPv4 header definition may contain a three-bit field, representing the three IPv4 flags, whose values are either one or zero. Alternatively, another program may contain a different format for the extraction of the same fields. The merged program produced by P4Visor will contain only one of these two definitions, with the other one being deleted. This means that only one of the input programs will be able to correctly reference the IPv4 flags, while the other program will reference a field that no longer exists in the merged program, resulting in an error. This indicates that under these circumstances, *the merged program produced by P4Visor will not be correct*.

Similarly, because P4 programs are usually [13] written by developers, there are no guarantees that identical structures will be given the same name across different programs. For example, a developer could define an Ethernet header, naming it `ethernet`, while, in a different P4 program, the same developer could assign to the same header structure (i.e., the same number of fields and the same width for each field) a different name (e.g., `ether`). In P4Visor, these two structures will not be considered equivalent just because of the different names and, as a consequence, will not be shared. It must be said that this last case does not affect *correctness* in the merged program, as the two input programs will use the different headers accordingly, rather it increases the total amount of resources used.

3.2.2 Naive Merge of Parse States

P4Visor's mechanism to merge parse graphs consists in sharing states that extract equivalent headers, using a custom parse state and custom header to "disambiguate and break conflicts in the merged parse graph" [8]. P4Visor determines the right program to process an incoming packet by verifying whether the packet has been encapsulated with the custom header or not. This can be better explained through the example in [8] reported in Figure 3.1, where the parse graphs of two programs (a) and (b) and the parse graph of the respective merged program generated by P4Visor (c) are illustrated.

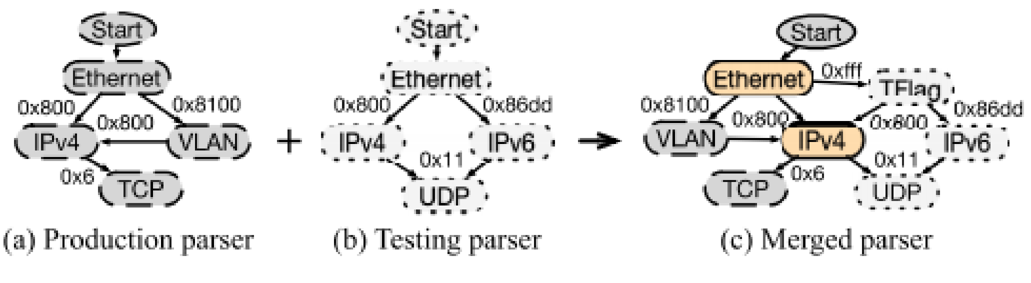


Figure 3.1: Example of a parse graph merge taken from [8]. In this example, a custom state, named `TFlag` is introduced to separate the states that are unique to each program, and equivalent states (i.e., Ethernet and IPv4) are shared.

In this example, an encapsulation mechanism is used to guarantee that, once merged, program (b) cannot transition to the `VLAN` state, and program (a) cannot transition to `IPv6` through the shared `ethernet` state. Specifically, P4Visor encapsulates packets meant to be processed by program (b), replacing the original ethernet types (e.g., `0x800, 0x8100`) for their custom `etherType` value, *0xfff*. The `ethernet` state in the merged parser transitions to a custom parse state, named `TFlag`, when selecting on this custom `etherType`. The *TFlag* state then extracts the custom header containing the original ethernet type of the packet (this field is copied from the original ethernet header), and it transitions only to states present in program (b)'s parse graph (i.e., `IPv4` and `IPv6`).

The above mechanism in P4Visor is not sufficient to guarantee the *isolation* of the input programs in the merged graph. This problem can be illustrated with the same example of Figure 3.1. Based on the way the parser is defined, the Production Program (a) may drop or process an incoming packet with the following protocol stack `Ethernet/IPv4/UDP` (e.g., it is not uncommon to process a packet that does not have the expected protocol stack). However, the Merged parser (c) now contains a conditional transition to UDP. Thus, a similar packet being received by the merged program would lead to a transition from the shared `IPv4` state to `UDP`, since no custom header is expected to be encapsulated for packets belonging to the Production program. This means that an input program would reach a state it would not have reached in its original parse graph, *which would represent a violation of the isolation property that the merging algorithm is expected to guarantee.*

Indeed, the implementation of P4Visor in [12] does not reflect the description of the mechanism to merge parse graphs described above. Equivalent parser states and headers are not shared. Rather, the states from one of the programs are only renamed (to prevent replacing the other program's states with equal names) by adding the *shadow_* prefix, and are then added to the merged program. This merge behavior is illustrated through the parse graph of the merged program obtained by P4Visor merging two identical programs in Figure 3.2. In the merged graph, the parse graphs of the two different programs are connected by the `shadow_parse_ethernet` parsing state (P4Visor requires all input programs to contain a `parse_ethernet` state) and the custom parse state is added to parse the custom header and disambiguate the programs. While this guarantees isolation, this solution is very inefficient, being effectively equivalent to the naive merge we present in Section 3.3.

3.2.3 Merging Multiple Programs

P4Visor was designed to provide a mechanism to test a new version of a P4 program. This is achieved by merging *two* P4 programs, the production program and the test program, into a single program, deploying it and comparing the output produced by processing the packets with each of the merged programs, in order to detect if the new version is producing the expected output.

Although this solution presents an innovative platform for testing new versions of network functionalities in large-scale networks, namely by facilitating the modular development of such

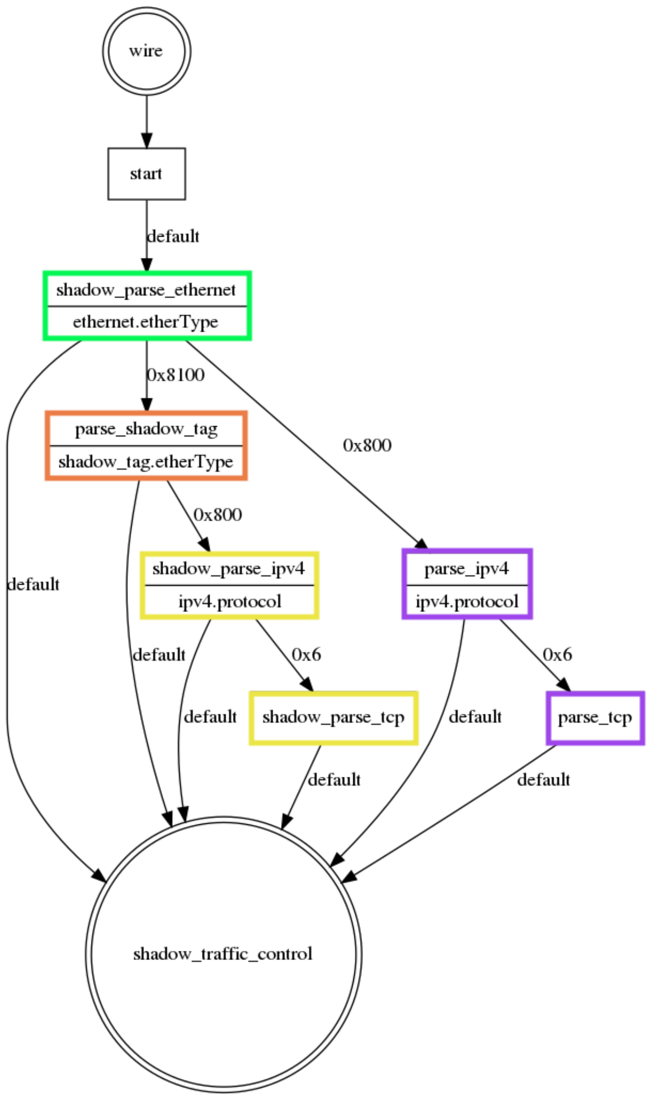


Figure 3.2: Resulting parse graph from the merge of two identical programs (with the same protocol stack) using P4Visor. Highlighted in green, the shared first state, that either contains the ethernet header belonging to the program highlighted in purple, or the custom header (highlighted in orange), that leads to the other program's graph, in yellow.

functionalities and by introducing a minimum resource-usage overhead, it restricts the number of programs that can be merged and consequently deployed to two, very similar programs. This restriction prevents this virtualization technique from achieving its potential as it (a) forces developers to write P4 programs that grow both in size and complexity as the number of functionalities increases, since only two programs can be merged, and (b) it prevents the modular development of a wide range of network functionalities, since each new functionality must be included in the two input P4 programs.

In summary, the actual P4Visor's merging mechanism produces merged parse graphs where *correctness* of the input programs is not guaranteed. Furthermore, equivalent states between parse graphs of the original programs are not shared. Hence, parser resources are not efficiently and correctly shared with this mechanism. Finally, P4visor targets a testing environment and is therefore not a generic virtualization solution as it only merges two similar programs.

3.3 System Overview

Our solution follows a Code Merging approach. A compiler receives multiple P4 programs (potentially developed by different tenants) as input, and produces, as output, a unique compiled representation of all the input programs which can potentially be deployed on a P4-programmable target, as illustrated in Figure 3.3. The produced representation (hereinafter referred to as the *merged program*) contains all of the necessary components to correctly apply the functionalities implemented in each input program (*functionally equivalent*), while simultaneously guaranteeing that those programs do not interfere with each other in the merged program (*isolation*).

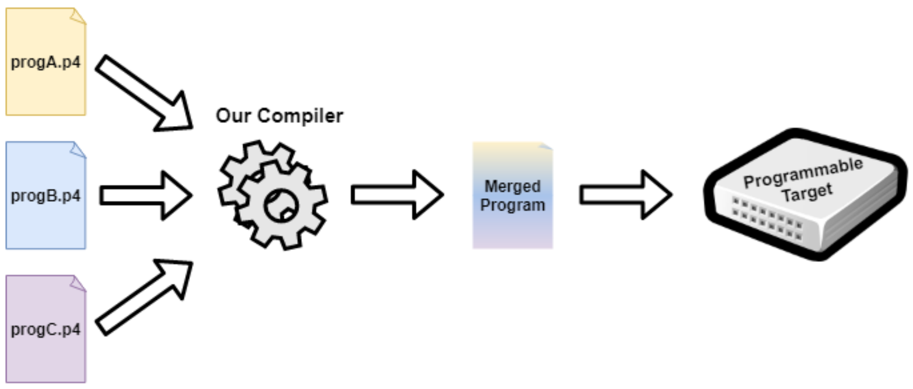


Figure 3.3: High-Level representation of our system. In this example, three P4 input programs are processed by our compiler and a merged program is produced.

In this work, we leverage an earlier version of the open-source P4 compiler, *p4c* [14], the one used by the P4Visor system [8] we build our solution upon. In a front-end compilation pass, *p4c* generates an HLIR of a P4 program. The HLIR is managed as a large Python object by the p4c front-end compiler. This object can then be translated to different compiled formats by different target-specific compiler back-ends. For example, our solution produces merged programs in JSON format, which allows us to program the P4 reference software switch, *bmv2* [15]. Working with the HLIR allows our system to merge the programs without modifying the original source files and to potentially produce an intermediate representation of a merged program which, fed to different back-ends, can produce different target-specific compiled versions.

Our compiler generates the HLIR objects for every input program and creates the final merged program by adding the elements from each program into a single HLIR object, which is finally translated into a JSON file and deployed on *bmv2* (for testing). To ensure that the original P4 programs are correctly represented in the merged program, and that the processing logic of each individual program is not affected by other programs, the different components from each program must be correctly integrated into that single HLIR with the dependencies between the different programs' objects being modified when necessary (e.g., an object that represents a parse state from a program has references to the states it reaches, and to the headers it extracts).

In addition to *functional equivalence* and *isolation* among the input programs, a code merging solution should also aim at reducing the number of resources collectively used by the merged programs. In practice, most P4 targets have strict restrictions on the amount of resources that can be used by the deployed program. This is particularly important if we consider that virtualization allows targets to run many different programs concurrently on the same programmable network device. For this reason, our system includes optimizations to reduce the amount of resources used by the merged program. At a high level, we achieve this goal by establishing equivalencies between resources that belong to different programs, enabling its sharing without compromising *isolation*, allowing for redundant components to be eliminated in the merged program. As a result, our solution reduces the overall resource usage, enabling the deployment of more programs than the alternatives.

In this work, we solely focus on the first stage of packet processing, namely packet parsing. For the other components of the data plane we leverage the P4Visor system. In a parser written in P4, received packets traverse a graph of user-defined parsing states. States are responsible to parse the packet bitstream in specific structures, named headers. Transitions between states represent conditional branching on protocol field values present in the packet (an example of a parser in P4 can be seen in Figure 2.2). Since P4 requires the developers to define every aspect of the packet processing behaviour, different programs commonly define different headers and parse graphs. Additionally, the internal structure of the header definition for the same protocol, that is, the number and width of its fields, may differ across different programs. For instance, a program that uses the flags present in a TCP header may declare them individually, that is, one field per flag, while another program that does not use them may declare a single field containing all of the flags. The absence of any standard definition of the protocol headers in P4 makes the process of finding and merging similarity across different programs challenging.

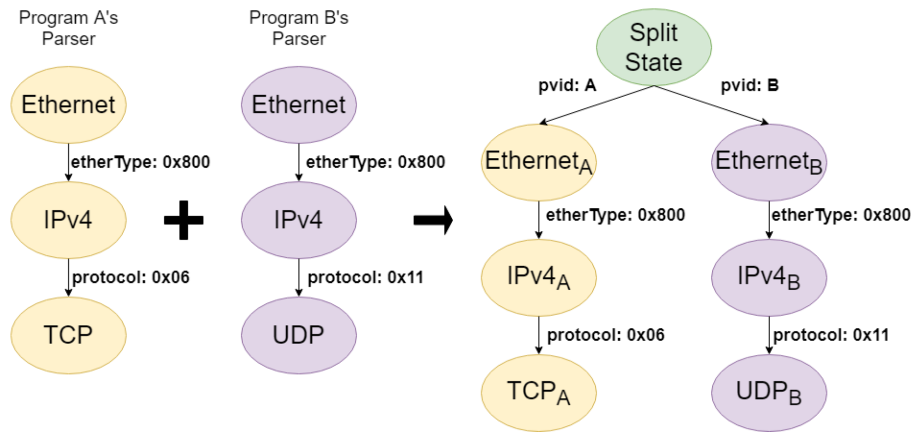


Figure 3.4: Naive parse graph merging process, introducing an additional state to combine multiple input graphs.

Therefore, to represent multiple P4 programs into a single merged program, it is necessary that the parse graphs from each input program are correctly incorporated in the merged program.

A naive approach to achieve this is illustrated in Figure 3.4 (this is similar to the approach adopted in P4Visor [8]). As it can be inferred from the figure, the parser graph of the merged program would contain multiple separate states (i.e., Ethernet and IPv4), implementing equivalent functionality. This choice is not optimal, since the presence of additional states and transitions require additional resources to store the parser graph, and those resources are scarce on most targets [3]. To improve this naive merge approach, our key idea is to establish equivalences (precisely defined later) between the headers and the related parser states across the input programs, with the goal to reduce the overall resources used by removing redundant components in the merged graph. In Figure 3.4 the two input parse graphs are entirely replicated in the merged program's parse graph. However, only the last states (TCP and UDP) are unique to each parse graph. Our compiler finds those equivalences and merges the remaining equivalent states, thus reducing the overall resources required to store the parse graph of the merged program, as can be seen in Figure 3.5. The challenge is to improve the efficiency of the merging process, while guaranteeing the *correctness* of each input program and the *isolation* between programs.

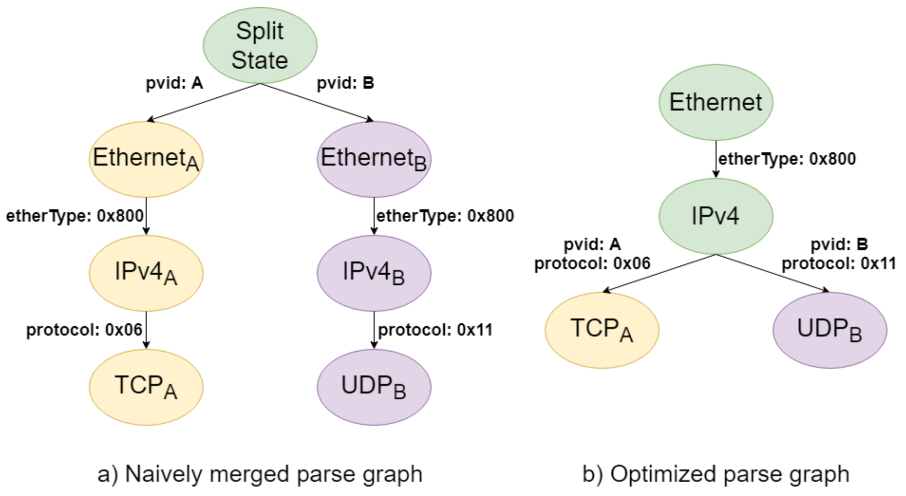
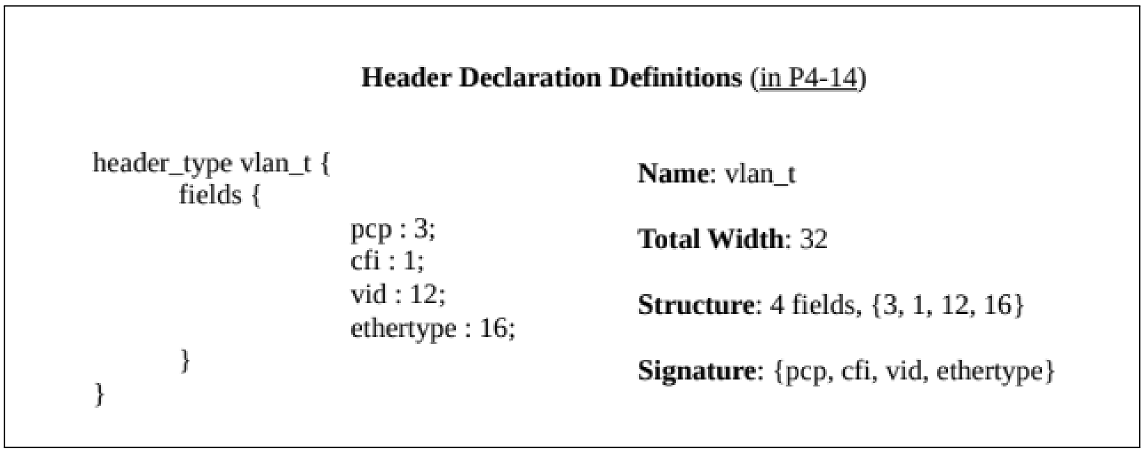


Figure 3.5: Reducing the size of the parse graph in the merged program by sharing *equivalent* states in the input programs. In a), we can see the result of naively merging the parse graphs, while in b) we see an example of a merge performed by our system, with shared states.

3.4 Merging Headers and Metadata

In order to correctly merge multiple P4 programs into a single program, one must guarantee that every header and metadata structure present in the input programs is properly represented in the merged program. However, it is possible to reduce the total number of headers present in the merged program by leveraging equivalences between header instances from the different input programs. Towards that goal, we have defined the four following degrees of equivalence which headers can exhibit between each other: *Strong Equivalence*, *Simple Equivalence*, *Weak Equivalence* or *No Equivalence*. We use an example of header definition in Figure 3.6 to introduce the terminology used to define the following header equivalences.



Header Declaration Definitions (in P4-14)

```
header_type vlan_t {
        fields {
                        pcp : 3;
                        cfi : 1;
                        vid : 12;
                        ethertype : 16;
        }
}
```

Name: vlan_t

Total Width: 32

Structure: 4 fields, {3, 1, 12, 16}

Signature: {pcp, cfi, vid, ethertype}

Figure 3.6: Definitions used to describe the different components of a header declaration. The *Name* represents the name given to this particular header. The *Total Width* represents the total number of bits extracted into the structure. The *Structure* represents both the number of fields and their individual widths. The *Signature* represents the names given to each field.

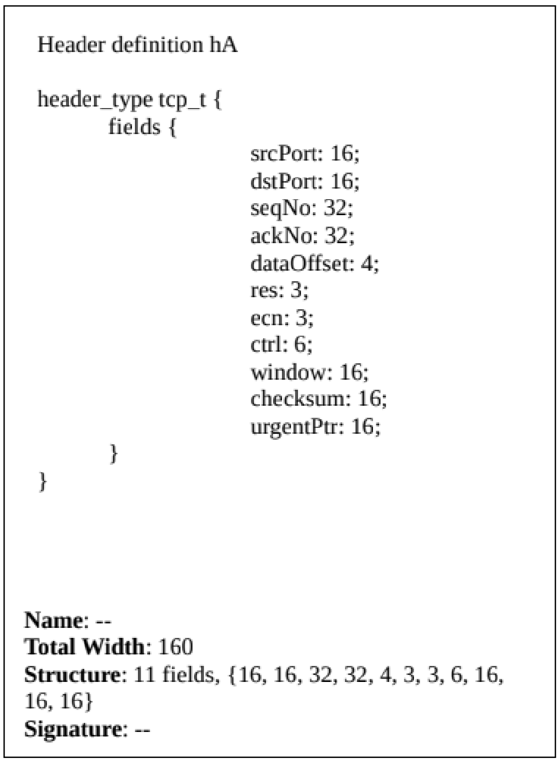
3.4.1 Header Equivalences

Header definitions for the same protocol in P4 may differ from each other in different ways. Since even minor differences may turn crucial for the correct merge of the programs, with the aim to guide the design of our merging algorithm, we have formally defined the differences between headers and classified the following four different cases:

- **Strong Equivalence:**
  - Two headers are considered *strongly equivalent* if and only if they have the same *name*, *total width*, *structure* and *signature*.
- **Simple Equivalence:**
  - Two headers are considered *simply equivalent* if and only if they have the same *total width* and *structure*, but have different *names* or *signatures*.
- **Weak Equivalence:**
  - Two headers are considered *weakly equivalent* if and only if they have the same *total width*, but a different *structure*.
- **No Equivalence:**
  - Two headers are considered *non-equivalent* if and only if they have different *total widths*.

3.4.2 Equivalence Implications

Depending on the type of equivalence established between two headers, different processing steps are required by our compiler to merge the parse states extracting those headers. In case a *Strong Equivalence* is established, as fields in the two headers are defined in the exact same way, the headers can be shared by only including one of them in the merged program. In case a *Simple Equivalence* is established, the headers can also be shared by only including one of them in the merged program, but because their name and/or signature is different, some renaming is required for one of the programs to match the header reported in the merged program. In case a *Weak Equivalence* is established, the two headers can still be shared by only including one of them in the merged program. However, because in this case the headers structure is also different the necessary renaming turns to be more complex than for the *Simple Equivalence* case.

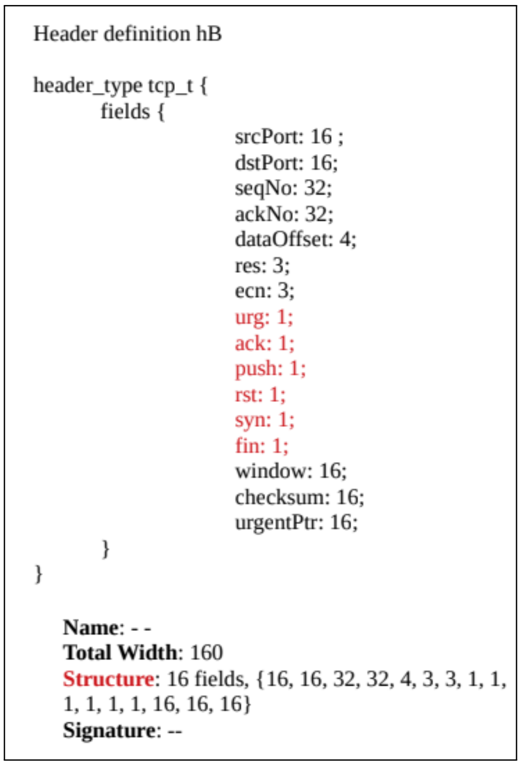


Header definition hA

```
header_type tcp_t {
        fields {
                        srcPort: 16;
                        dstPort: 16;
                        seqNo: 32;
                        ackNo: 32;
                        dataOffset: 4;
                        res: 3;
                        ecn: 3;
                        ctrl: 6;
                        window: 16;
                        checksum: 16;
                        urgentPtr: 16;
        }
}
```

Name: --
Total Width: 160
Structure: 11 fields, {16, 16, 32, 32, 4, 3, 3, 6, 16, 16, 16}
Signature: --

(a)



Header definition hB

```
header_type tcp_t {
        fields {
                        srcPort: 16 ;
                        dstPort: 16;
                        seqNo: 32;
                        ackNo: 32;
                        dataOffset: 4;
                        res: 3;
                        ecn: 3;
                        urg: 1;
                        ack: 1;
                        push: 1;
                        rst: 1;
                        syn: 1;
                        fin: 1;
                        window: 16;
                        checksum: 16;
                        urgentPtr: 16;
        }
}
```

Name: - -
Total Width: 160
Structure: 16 fields, {16, 16, 32, 32, 4, 3, 3, 1, 1, 1, 1, 1, 1, 16, 16, 16}
Signature: --

(b)

Figure 3.7: Example of two Weakly Equivalent headers, with the differences in their structure highlighted in red.

In summary, among the different types of equivalence described above, *weak equivalence* represents the one that requires the most careful treatment to perform a correct merge of the related headers. To better illustrate that case, we compare the definitions of two *weakly equivalent* headers side by side in Figure 3.7. The two header definitions h_A (a) and h_B (b) for the TCP protocol in Figure 3.7 have a different structure. More precisely, h_B replaces the six-bit `ctrl` field in h_A, which is meant to extract the TCP flags, by six individual one-bit fields. Although the structure of these two headers is slightly different, it is still relatively clear that these two headers still represent the same protocol and could potentially be merged. However, the following considerations must be taken for these headers to be merged. More specifically, the coarser-grained fields are maintained and the finer-grained are properly translated. For example, as the `ctrl` field is more coarser-grained in header h_A, we kept h_A on a single merged header. However, in the merged result, we must ensure that any reference to the h_B fields in the original program is properly translated to the header definition included in the merged program. Hence, our compiler translates any reference to those missing fields into the corresponding bits of the header h_A included in the merged program. Thus, for example, a reference to the `ack` field is replaced by a reference to the second bit of the `ctrl` field.

3.4.3 Programs' Metadata and P4Visor++ Custom Header

An analogous principle is also applied to the user-defined metadata structures potentially present in the input programs, although those structures are merged separately from headers. Besides, every P4 program contains *Standard Intrinsic Metadata*, a metadata structure associated with the operation of the target, providing various information regarding the received packet, such as the ingress port. Because those metadata are not user-defined, our solution ignores the multiple instances present across the input programs, and considers only one instance of this metadata in the merged program.
Additionally, the merged program also contains our custom header, which is used to infer the right input program to process an incoming packet once the merged program is deployed. Our custom header (called `upvn`) comprises a four-bit field (`pvid`) containing a unique ID assigned to the original program. This header cannot be shared with any of the headers from the input programs. The purpose and use of this header will be explained in the following section.

3.5 Merging Parse States

Parse states are meant to extract protocol headers in received packets. Each parse state can be defined to extract one or more headers. Transitions to the next states in the graph are encoded by matching values present in extracted header fields and/or metadata against specific values (e.g., an IPv4 parse state will match the `protocol` field present in the IPv4 header, and transition to the TCP state if that value is 6). Additionally, parse states can also modify packet metadata.
Our algorithm leverages equivalences between parse states in order to merge parse graphs from different programs. In case such an equivalence is established, our algorithm shares the equivalent states, applying the necessary modifications in order to preserve both *isolation* and *correctness*.
As first state of the merged parse graph, our algorithm includes a custom parse state, named `parse_upvn`. This state extracts our custom header containing the program ID in the `pvid`. This state serves as the entry point for every program in our merged program. In fact, as programs are added to the merged program, transitions are added to this state, pointing to the first state of each program, whether that is shared or not. This mechanism was illustrated in Section 3.3, in Figure 3.5.

3.5.1 Equivalence Between Parse States

A parse state must meet certain criteria to be considered equivalent to another state, and thus possibly being shared in our algorithm. Equivalent states must (i) extract equivalent headers (any type of equivalence), (ii) not modify metadata possibly used by other programs (e.g., standard intrinsic metadata exposed by a target architecture), and (iii) be in compatible topological levels. The pseudo-code verifying these three criteria is summarized in Algorithm 1.

Algorithm 1 Determining if two parse states can be shared

```
1: Input
2:     P1   Merged program
3:     P2   Program being added
4:     S2   Parse state from P2 being added
5: procedure SHARE_STATE($P1, P2, S2$)
6:     $H2 \leftarrow get\_extracted\_header(S2)$
7:     $S1 \leftarrow get\_state\_extracts\_equivalent(P1, H2)$
8:     if $!extracts\_same\_headers(S1, S2)$ then
9:         $P1.add(S2)$
10:        return
11:    end if
12:    if $uses\_same\_metadata(P1, S2)$ then
13:        $P1.add(S2)$
14:        return
15:    end if
16:    if $!compatible\_topo\_level(P1, P2, S1, S2)$ then
17:        $P1.add(S2)$
18:        return
19:    end if
20:    $P1.current\_topo\_level = S1.topo\_level$
21:    $P2.current\_topo\_level = S2.topo\_level$
22:    $modify\_merged\_state(P1, S1, S2)$
23: end procedure
```

As it can be seen in the pseudo-code, to assess whether a parse state $S2$ in program2 ($P2$) can be shared, our algorithm first seeks a candidate parse state $S1$ in program1 ($P1$) by looking at the first header that the former extracts, and looking for a state in the merged program that extracts an equivalent header (lines 6 and 7). Then, the algorithm verifies whether the three conditions above hold for the pair of states $S1, S2$. If any of these condition is not met, the state cannot be shared and is therefore added to the merged program (lines 9, 13 and 17), and the algorithm continues with the next state. The first test (line 8) verifies whether the states $S2$ and $S1$ extract equivalent headers in the same order. The second test (line 12) verifies whether the state $S2$ modifies any metadata which is used by any of the programs already added to the merged program. This check is meant to avoid that different programs sharing this state will modify the same metadata with different values risking to write unexpected values for some of the merged input programs. Finally, the third test verifies whether the states $S2$ and $S1$ are in compatible topological levels (line 16).

Maintaining a correct topological order between the programs is an important requirement in our algorithm, because it prevents cycles from being created in the merged program. We explain this property in further detail in the following section.

3.5.2 Topological Levels

In our merge algorithm, states in the parse graph of each program are assigned a topological level. This level represents the position of a state relatively to the other states in the same parse graph, providing information about the correct order in which headers must be extracted. A state will be always assigned a level greater than the states preceding it, and lower than the states succeeding it. Each state is assigned a level corresponding to the longest possible path on the parse graph to it. As shown in Figure 3.8, state `IPv4` is reached directly from `Eth`, but it is not assigned the

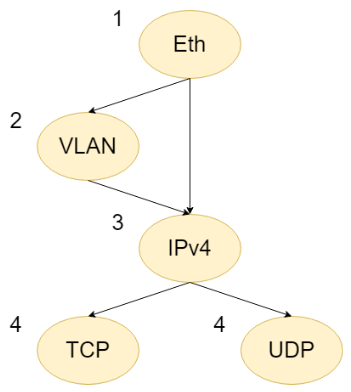


Figure 3.8: Example of the topological level attribution in a parse graph.

level 2 (which would be Ethernet state's level plus one), since it is also reached through the `VLAN` state. In the case of `TCP` and `UDP`, both states are in the same level, as they are reached only by the `IPv4` state. Before starting to add the states to the merged program, our algorithm sorts the states by their topological level, forcing them to be added in the same order in which they are extracted in their original graphs.

This mechanism is introduced to help detect states that are considered equivalent but do not represent the same network protocol layer, and prevent those from being shared. This mechanism is based on the assumption that standardized protocols are extracted in the same order across different programs. For example, IPv4 headers are expected to be extracted after the Ethernet header, while VLAN headers are expected to be extracted after the Ethernet header and before the IPv4 header across all input parse graphs. Using this logic, any header extracted, for example, after the IPv4 header in an input program, expects to find an equivalent instance in any other input program where such header is not extracted before IPv4. Any case where this does not occur indicates that these headers, although equivalent, may not represent the same network layer, and sharing them can result in a loss of efficiency of our merging process (e.g., transitions within a shared state are less likely to be equivalent across the programs if the headers extracted do not represent the same networking layer).

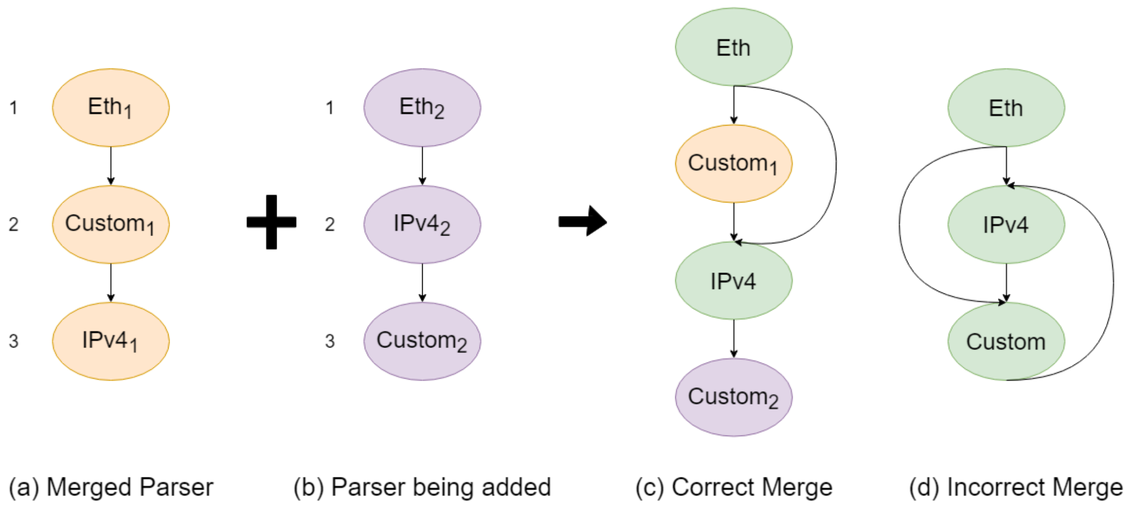


Figure 3.9: Example of a parse graph (b) being added to the merged program (a). In case (c), the merge is topologically correct, while in (d) it is incorrect, as a cycle is introduced.

The concept of topological levels is illustrated through the merge examples in Figure 3.9. In the example, states with the same name (in the figure) from both programs are candidate states to be shared (i.e., Eth$_1$ is a candidate for merging with Eth$_2$, IPv4$_1$ with IPv4$_2$ and Custom$_1$ with Custom$_2$). Before our algorithm starts checking for equivalences between states, each state (from both graphs) is assigned a topological level (this is represented, in the figure, by the number visualized at the left side of each state). When our algorithm attempts to add the first state from parser (b), the procedure previously described in Algorithm 1 is followed. In this case, Eth$_2$ is equivalent to Eth$_1$ and so these states are shared, with both programs updating their current topological level (lines 20 and 21). The program's current topological level represents the level of the last state to be added for each program, so both programs will be at level 1 after sharing states Eth$_1$ and Eth$_2$. At the next iteration, the algorithm attempts to add IPv4$_2$, identifying state IPv4$_1$ as a candidate to be shared. This time, the topological level of the states is different (3 for the merged program, and 2 for the program being added). However, the merge still occurs, as our algorithm only enforces that the levels of each state are greater or equal than the last state that was shared. After the merge of the states $\{IPv4_1, IPv4_2\}$, each program takes the level of the last shared state as their current level, and so program (a) moves to level 3, while (b) moves to level 2. Finally, when attempting to share state Custom$_2$ with state Custom$_1$, this condition does not hold since the topological level of program (a) is greater than the level of its state (program (a) is at level 3, while Custom$_1$ is at level 2). For this reason, our algorithm decides that states Custom$_2$ and Custom$_1$ cannot be shared, and Custom$_2$ is added to the merged program. This prevents the outcome seen in (d), by placing both Custom$_1$ and Custom$_2$ in their topologically correct positions, that is, Custom$_1$ before the shared IPv4 state, and Custom$_2$ after the shared IPv4 state. The resulting graph is the one reported in (c).

If, however, the order of the input programs is swapped, that is, the parser from (b) is added to (a), a different merged graph is produced, as shown in Figure 3.10. In this case, the states

Custom_1 and Custom_2 are shared, since, this time, the program being added contains its Custom state at a lower topological level than the IPv4 state. This example highlights the fact that, under certain circumstances, changing the order in which programs are added produces different resource sharing results with our algorithm. As a consequence, we have included some pre-processing stages in our algorithm with the aim to explore multiple permutations of the sequence of the input programs to obtain better optimization results.

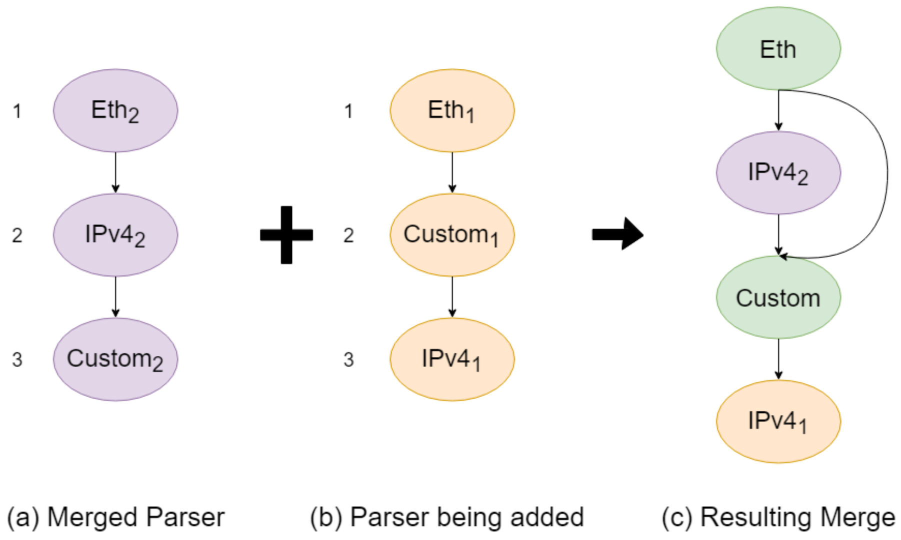


Figure 3.10: Result of merging the programs in a different order.

Additionally, a different problem can occur when merging graphs that contain multiple states at the same topological level. When merging the graphs, the states are sorted based on their topological levels, and added from lowest to highest level. However, when multiple states are at the same level, an extra processing step is required to decide the order in which they must be added. The necessity for this mechanism is made clearer with the example illustrated in Figure 3.11.

In this figure we assume that *Added Program* is being added to *Base Program*, and that states named with the same letter are equivalent. In this scenario, at the topological level 3, we could merge state B from *Added Program* with B from *Base Program*. Once two states are shared, the program's current topological level is modified, identifying the topological level of the last shared state. In this case, *Added Program* would be at level 3 and the *Base Program* would be at level 2. Then, state C from *Added Program* would be shared with state C from *Base Program*, and the topological level would remain at 3 for *Added Program*, but move to 3 for *Base Program*. This would prevent the E states from being subsequently shared, since in the *Base Program*, this state has level 2, and according to our algorithm, a state cannot be shared if its level is inferior to the current program's topological level. However, if the C states are shared only after states B and E have been shared, then all of the equivalent states can be shared. To prevent the first condition from happening, we sort states at the same topological level in the *Added Program* program based on the topological level of the equivalent states in the *Base Program*. Intuitively, we give priority

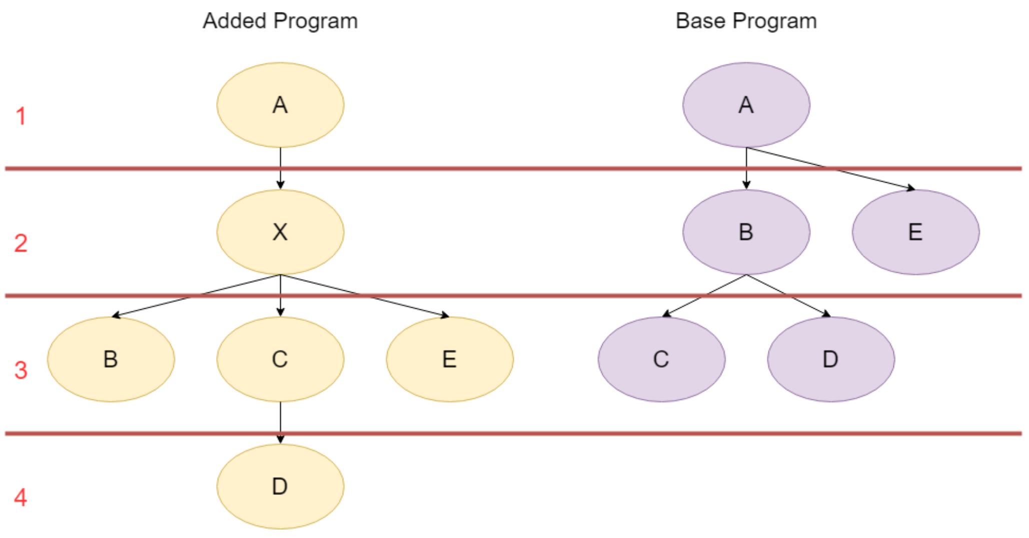


Figure 3.11: Example of two parse graphs where the order in which states at the same topological level are added impacts the merged parse graph.

to states that are equivalent to states that are at a lower topological level. This guarantees that the problem described above does not occur, since we force our sharing mechanism to move the *Base Program*'s current topological level from lower to greater, while maintaining the same level in the *Added Program*.

3.5.3 Sharing Equivalent States

Once an equivalence between two states has been established, those states can be shared. Our algorithm allows for states that select on distinct matching fields and transition to different states to be shared. This is achieved by modifying accordingly the select statement and the transitions present in the merged program's related state. These changes are performed, however, preserving *isolation* and *correctness*.

In order to guarantee these two properties, our algorithm must ensure that the possible paths of any input program's parse graph are maintained after the merge occurs (*correctness*) and that each program only traverses states in the merged graph which are equivalent to the ones in its original parse graph (*isolation*). For this purpose, our algorithm modifies states in the merged program to additionally select on the program ID field, `pvid`, of our custom header. The P4 language allows for multiple fields to be selected on, and so our algorithm adds the `pvid` field to the fields already present in the select statements of the merged program, only where necessary, to disambiguate the merged programs.

Equivalent states that select on different fields can be shared too, by selecting on the union of respective fields (e.g., state A selects on field a, state B selects on field b. The merged state AB will select on a + b). This technique allows our algorithm to increase the total number of shared resources in the merged program, but it requires ignoring fields in select entries for states which originally did not select on those fields.

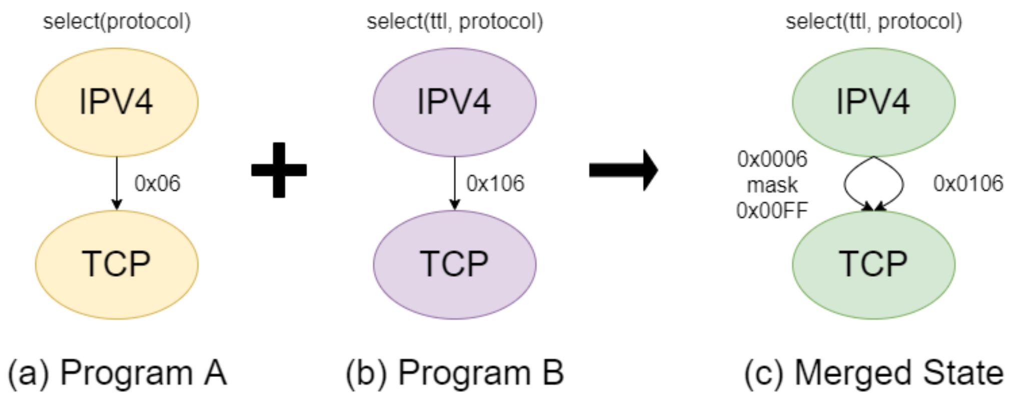


Figure 3.12: Example of two equivalent states with different select statements being shared

To rewrite accordingly the entries in the select statements of the merged program, we leverage the P4's `masking` operation to specify which bits are relevant for each select entry. Consider the two programs, A and B, in Figure 3.12, both containing an IPv4 parse state. In program A, that state selects on IPv4's field `protocol`, and transitions to the TCP parse state for a protocol value of `0x06`. In B, the equivalent state selects on two IPv4 fields instead, `ttl` and `protocol`, transitioning to the TCP state in case the combined values are `0x0106` (with a `ttl` value of 1 and a `protocol` value of 6). When these two IPv4 states are shared, the resulting select statement will be in the format `select(ttl,protocol)` (the program ID is not included for sake of clarity in this example). For program B, the transition that must be present in the merged program is the same as the original one specified in program B, as the select statement is the same. However, for program's A transition to TCP the IPv4 *ttl* field must be masked.

Algorithm 2 Adding different select fields to the state in the merged program

```
1: Input
2:     S1   State from the merged program
3:     S2   State from P2 being added
4: procedure ADD_DIFFERENT_SELECT_FIELDS($S1, S2$)
5:     $new\_fields \leftarrow get\_new\_fields(S1, S2)$
6:     $S1.select\_statement.append(new\_fields)$
7:     for $(key, mask), next\_state\ in\ S1.transitions$ do
8:         for $field\ in\ new\_fields$ do
9:             $key = key + "0" * field.width$
10:            $mask = mask + "0" * field.width$
11:        end for
12:    end for
13: end procedure
```

For example, when selecting on two 4-bit long fields, masking the select entry with `11110000` will result in the state only looking at the first four '1' bits of the given value (while bits set to 0 are ignored). Following the example, program A's transition in the merged program would look like this : `0x0006 mask 0x00ff : tcp_state` (both `ttl` and `protocol` have 8 bits).

Algorithm 3 Modifying the transitions of the state being added

```
1: Input
2:     S1   State from the merged program
3:     S2   State from P2 being added
4: procedure MODIFY_ADDED_STATE($S1, S2$)
5:     $map\_f \leftarrow Dict()$
6:     $index \leftarrow 0$
7:     for $f2$ $in$ $S2.select\_statement$ do
8:         $map\_f[f2] = index$
9:         $index = index + f2.width$
10:    end for
11:    for $(key, mask), next\_state$ $in$ $S2.transitions$ do
12:        $new\_key = ""$
13:        $new\_mask = ""$
14:        for $f1$ $in$ $S1.select\_statement$ do
15:            if $f1$ $in$ $S2.select\_statement$ then
16:                $index = map\_f[f1]$
17:                $new\_key + = key[index : index + f1.width]$
18:                $new\_mask + = "1" * f1.width$
19:            else
20:                $new\_key + = "0" * f1.width$
21:                $new\_mask + = "0" * f1.width$
22:            end if
23:        end for
24:    end for
25: end procedure
```

In Algorithm 2, the process to introduce new fields to the shared state is illustrated. In line 5, we place in `new_fields` every field from S2 that does not exist in S1. These fields are then appended to the end of the select statement (line 6), and each transition is modified to ignore the newly added fields, by adding a number of zeros equal to the number of added bits to both the matching value and the mask (lines 9 and 10), thus ignoring the fields.

Then, as illustrated in Algorithm 3, the transitions of the state being added are also modified prior to being introduced in the shared state. In lines 8 and 9, the selected fields are mapped to their exact position in the bits being selected. This is required since the order in which fields are selected in the shared state may not be the same as the order present in the state being added. To modify the transitions in a way that they are prepared to be added to the shared state, the fields selected in the shared state are traversed (line 14), and the transition is created by either retrieving the bits relative to the equivalent field in S2, if such exists (lines 15 through 18), or by ignoring the field, in case the field from the shared state does not exist in the state being added (lines 19 through 21).

3.6 Optimizing Transitions in the Merged Graph

Once every input program has been added to the merged program, the resulting parse graph contains every state present in the original parse graphs, with some of the equivalent states being shared. However, every transition from the input programs is present in the merged parse graph, as the introduction of the program ID duplicates transitions that were equivalent among original programs into unique entries. Indeed, some of these transitions can also be merged and, as a result, the total number of transitions in the parse graph of the merged program reduced.

Towards that goal, we must first identify the cases in which transitions are redundant and can

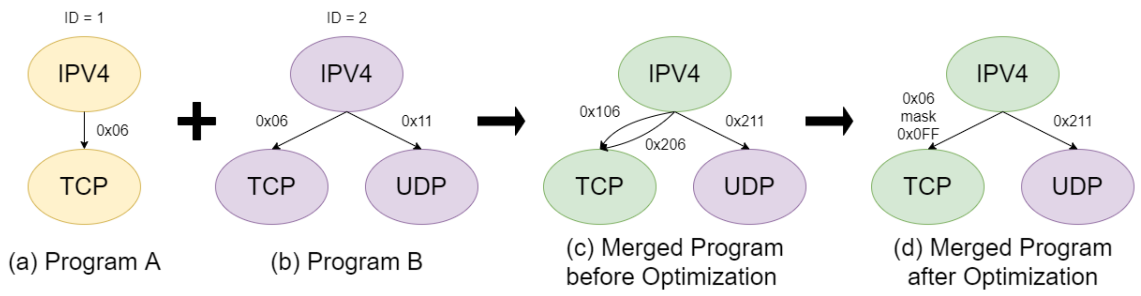


Figure 3.13: Example of the optimization algorithm reducing the number of transitions in a shared state

therefore be merged, if and only if that does not violate the *isolation* of the merged programs. The merge of these redundant transitions is possible only when every program sharing a state contains the same transition value and the same following state. In this case, there is no need to leverage the program ID to specify the programs which can perform a certain transition. This case is better depicted through an example, in Figure 3.13, where a transition to the TCP state is removed. This optimization can occur because both programs contain a transition to the TCP parse state (the TCP state is shared in the merged program) and with the same value. Since only these two programs share this state, a packet arriving at this state with a *protocol* value of `0x06` is always allowed to transition to the TCP state, regardless of what program is expected to process the packet at the moment. Hence, the two transitions can be represented by a single entry.

Differently, in the same example of Figure 3.13, the transition to the UDP state must check the program's ID to guarantee *isolation*. Program A must not transition to UDP upon receiving a packet with a `protocol` value of `0x11`, as that was not expressed in its original parse graph (a). The optimization described above could not occur for any additional program sharing the IPv4 state but not containing a transition to TCP. To formalize the conditions enabling this optimization, we consider a set of transitions within a state to be optimized into a single transition if and only if all the following criteria are met:

- the number of transitions with the same transition value (ignoring the program IDs) is equal to the number of programs sharing the state.
- the next state for each of those transitions is the same.

The pseudo-code for this optimization is reported in Algorithm 4. As a first step, we find the number of programs that are sharing a state by counting the number of unique program IDs present in the transitions. We also create a copy of the transitions (`transitions_copy`), and delete redundant transitions if they exist. This copy will, at the end of the process, replace the original transition dictionary. Additionally, we verify if every transition is present in each input program (`remove_id`), that is, if every transition can be optimized, and eliminate our custom select field from the select statement, reducing the amount of memory used by the state.

Algorithm 4 Optimize transitions in shared states

```
Input
    S1    State being optimized
procedure OPTIMIZE_STATE($S1$)
    $n \leftarrow get\_programs\_sharing\_state(S1)$        ▷ # of programs sharing the state
    $transitions\_copy \leftarrow S1.transitions$
    $remove\_id \leftarrow True$
    for $transition\ in\ S1.transitions$ do
        if $transition\ in\ transitions\_copy$ then
            $similar\_transitions \leftarrow get\_similar\_transitions(S1, transition)$
            if $similar\_transitions.length() == n - 1$ then
                $transitions\_copy.pop(similar\_transitions)$
            else $remove\_id \leftarrow False$
            end if
        end if
    end for
    $S1.transitions \leftarrow transitions\_copy$
    if $remove\_id$ then
        $S1.select\_statement.remove(pvid)$
    end if
end procedure
```

3.7 Summary

In this chapter, we presented the design of our solution to correctly and efficiently merge the parse graphs of multiple P4 programs. Firstly, we enumerated the requirements of our solution, in Section 3.1. Then, we presented a summary of the limitations of the state-of-the-art Code Merging solution (P4Visor), in Section 3.2, followed by a high-level description of our system, in Section 3.3. Then, we specified the criteria used to establish equivalence between headers and how they are added to the merged program, in Section 3.4. Afterwards, we described how the merging of parse states is achieved, by specifying how states are shared and added to the merged program, in Section 3.5. To finalize, in Section 3.6, we presented an optimization that allows our algorithm to reduce the total number of transitions in the merged parse graph.

Chapter 4

Implementation

In this chapter we describe the implementation of P4Visor++, detailing the main system's components developed during the course of our work. Our work leverages the P4Visor's code base, which is available as open-source software under Apache License on github. More in detail, we have modified P4Visor's code base to integrate several mechanisms, namely, to share headers, to merge parse graphs and to work with a number of input programs greater than two.

In Section 4.1, we present our system's structure, describing at a high level how the P4Visor code base has been modified. In Section 4.2, we describe how our custom header and parse state are declared and integrated into the merged program's intermediate representation. In Section 4.3, we detail how equivalencies between headers and metadata structures are established, and in Section 4.4 we explain how equivalencies between parse states are determined, and how they are leveraged to merge those states. To conclude, in Section 4.5, we present how the total resource usage of the computed merged program is further optimized by removing duplicate transitions in the merged parse graph.

4.1 System Structure

To merge different input P4 programs, our system leverages the High-level Intermediate Representation, or HLIR [16], of a P4 program. As illustrated in the previous chapter, the HLIR can be obtained as a python object by the front-end pass of the P4 reference compiler [14]. This python object stores the different elements of a P4 program across different dictionaries, as illustrated in Listing 4.1.

The names of the elements in the program are used as keys to index the HLIR dictionaries (e.g., the object that represents the `parse_ethernet` state of a program can be retrieved with the syntax `hlir.p4_parse_states['parse_ethernet']`). The objects stored in these dictionaries, that are program elements, are also often linked with each other, as is the case with parse state objects which, in turn, contain a dictionary called `state.branch_to` listing all the next states which can be reached from the current state.

The open-source P4 compiler p4c [14] contains a python script, named `__main__.py`, that receives a P4 program as input and produces its HLIR, that is then used by a second script, `gen_-`

`json.py`, to generate the JSON file that can be installed on the *bmv2* software switch target. This process can be seen in Figure 4.1a.

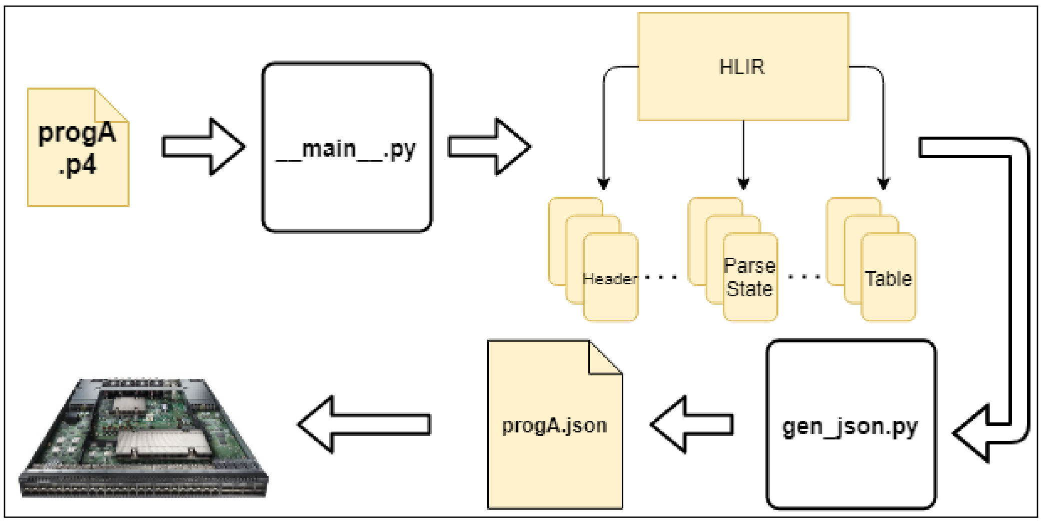
Listing 4.1: Organization of the P4 program's elements inside a HLIR python object.

```
self.primitives = []
self.p4_objects = []
self.p4_primitives_ = OrderedDict()
self.p4_actions = OrderedDict()
self.p4_control_flows = OrderedDict()
self.p4_headers = OrderedDict()
self.p4_header_instances = OrderedDict()
self.p4_fields = OrderedDict()
self.p4_field_lists = OrderedDict()
self.p4_field_list_calculations = OrderedDict()
self.p4_parser_exceptions = OrderedDict()
self.p4_parse_value_sets = OrderedDict()
self.p4_parse_states = OrderedDict()
self.p4_counters = OrderedDict()
self.p4_meters = OrderedDict()
self.p4_registers = OrderedDict()
self.p4_nodes = OrderedDict()
self.p4_tables = OrderedDict()
self.p4_action_profiles = OrderedDict()
self.p4_action_selectors = OrderedDict()
self.p4_conditional_nodes = OrderedDict()
```

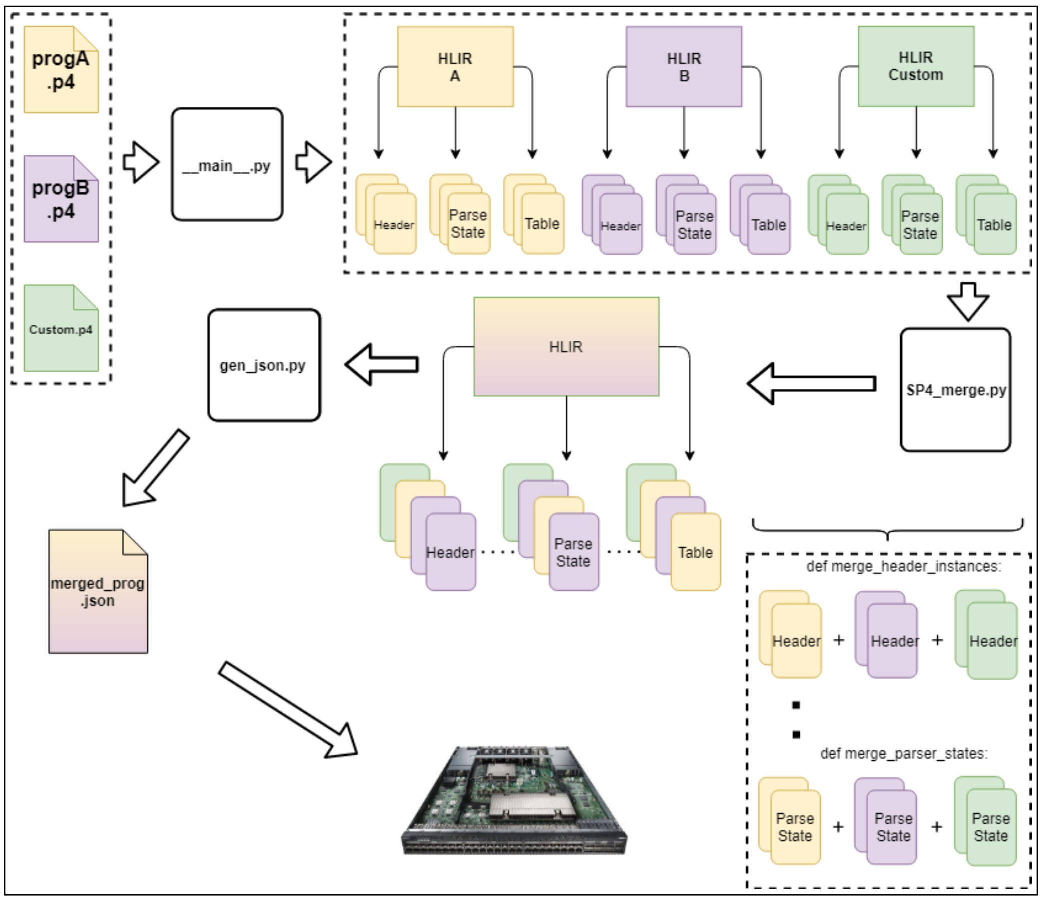
To merge multiple programs, P4Visor has modified p4c to receive two input P4 programs (rather than just one), and to create the respective HLIR objects. At this stage, P4Visor also generates the HLIR for an additional program used to introduce custom program elements in its merge process. The three HLIR objects are subsequently passed to `SP4_merge.py`, a script that performs the merge of the two input programs and outputs a merged HLIR. `SP4_merge.py` creates an empty HLIR object, fills its dictionaries with the content of one of the input programs and of the custom program. Once these elements have been integrated into the new HLIR object, `SP4_merge.py` starts the real merge process by adding program elements from the HLIR of the second input program. An illustration of this process is provided in Figure 4.1b.

`SP4_merge.py` consists of a series of functions that are sequentially executed, each one responsible for merging different types of elements from two input programs into a single HLIR object. Except for a few program elements, this process consists of updating the dictionaries of the merged HLIR with the content of the second program's dictionaries.

In the original P4Visor's execution, two functions are executed to merge the headers and the parsers from two input programs, namely `merge_header_instances` and `merge_parser_-states`. The former function performs the merge of the headers from the three programs, that is, the two input programs and the additional P4Visor-specific P4 program. It updates the headers dictionary in the merged HLIR with the content present in the respective dictionaries from the input programs, replacing existing elements with equal names. The latter function updates the



(a) p4c [14] workflow.



(b) P4Visor's workflow.

Figure 4.1: Comparison between p4c and P4Visor's workflow. The __main__.py script initiates the process by generating the HLIR objects of each input program. Then, p4c generates the JSON file, while P4Visor introduces SP4_merge.py, a script that receives the HLIRs as input and returns a merged HLIR as output. Only then, is gen_json.py used to generate the deployable program, in JSON format.

parse states' dictionary with the states in a similar manner. It first, however, renames the states already added to the merged program, preventing states from being replaced, as it occurs with headers. Once every state is present in the merged HLIR, the two programs parse graphs are linked through their first state (which must always be `parse_ethernet`), using for that purpose an ad-hoc parse state from the custom P4 program provided as a third input to `SP4_merge.py`.

We have mostly re-worked the above-described functions of the P4Visor's `SP4_merge.py` script to implement our solution. In detail, we have included mechanisms to (a) detect equivalent headers and parse states, (b) merge those components, sharing them when possible and (c) optimize the merged parse graph to further reduce resource usage. Additionally, we have modified `__main.py__` to receive and merge more than two P4 programs as input, and we have modified the custom P4Visor's P4 program to include additional program elements. We have also modified the HLIR object (`main.py` and `p4_parser.py`) to contain attributes that are useful in the merging process, such has topological levels for both the HLIR and its parse states, and equivalence lists that are auxiliary to the header merging mechanism. These elements are further explained throughout this chapter.

4.2 Custom Header and Parse State

In order to guarantee *isolation*, our solution introduces the extraction of a custom header through an ad-hoc parse state in the merged program. Our custom header, named *upvn*, was designed to be a small header (4 bits), containing only a program identifier to be leveraged in the processing of packets by the merged program to differentiate among the several input programs. For the purpose of our evaluation, we carry some additional information, namely the expected traversed path in the parse graph, encoded as a bitmap within our custom header through a special field called *p_map*. The *p_map* field is later compared with an analogous metadata field, set as the packet is parsed by the merged program (this mechanism will be fully explained in the next chapter presenting the evaluation of this work). The structure of this custom header is reported in Listing 4.2.

Listing 4.2: Header type definition for our custom header.

```
header_type upvn_t {
    fields {
        pvid : 4;
        p_map : 16;
    }
}
```

Each packet for our merged program must be prepended by our custom header, because the program ID is the first information required by the parser in the merged program to correctly steer the packet along the correct path in the merged parse graph. To achieve this goal, a state responsible for extracting the custom header and consequently selecting on the `pvid` field has been added. In Listing 4.3, we see how this state is introduced in the merged program's graph. `h_mg` is a reference to the merged program, that at this stage only contains states from one of the input programs, and the custom parse state. Our custom state is placed between the start state, an initial state present in all programs and the next state it transitions to, by replacing transitions accordingly (line 6 and line 14). Afterward, we replace every transition to the input program's first table in the `Ingress` stage with an equivalent transition (line 17) to a custom table introduced by us. This table replaces the Ingress stage of the input programs for a simple forwarding mechanism used for testing purposes.

Listing 4.3: Introducing the custom parse state into the base program.

```
1  #get the first state after start
2  first_state = h_mg.p4_parse_states['start'].branch_to[default]
3
4  #transition from start to custom state instead
5  h_mg.p4_parse_states['start'].branch_to.clear()
6  h_mg.p4_parse_states['start'].branch_to[default] =
7                                  h_mg.p4_parse_states['parse_upvn']
8
9  #add transitions to upvn state, e.g.,
10 #select(upvn.pvid)
11 #default : parse_ethernet
12 tempDict = OrderedDict()
13 tempDict[default] = first_state
14 h_mg.p4_parse_states['parse_upvn'].branch_to = tempDict
15
16 #change Ingress pointer to our custom Ingress table
17 set_parser_default_table_STC(h_mg.p4_parse_states, h_mg)
18
19 #add pvid to the select of each state, also modifying transitions
20 add_shadow_field_to_states(h_mg)
```

It is worth noting that this code is only applied in case the `start` state transitions directly to the first state (in most programs, this is a default transition to the ethernet parse state). If, however, the `start` state contains a conditional transition, that is, the state selects on a portion of the header before anything is extracted, using the field reference `current(..)` which allows the state to match on bits from the packet without extracting them, a different mechanism must take place. In this case, that select statement must also be present in our custom state, and each transition from the remaining input programs must reflect this change, by also matching on the bits that are looked at with `current(..)` and ignoring them using a mask.

In the case where every start transition is the same across the input programs (i.e., default transition to a shared state), the custom state's operation can be reduced to the extraction of the custom header, as *isolation* is already guaranteed by the fact every program transitions to the same state. Therefore, after the programs are merged and the transitions have been optimized, the `parse_upvn` state can be removed, with the custom header being extracted at the beginning of that state (lines 15 to line 19 of Listing 4.4). This deletion results in the reduction of the number of states and transitions by one.

Listing 4.4: Removing the custom parse state from the merged program.

```
if len(h_mg.p4_parse_states['parse_upvn'].branch_to.items()) == 1:
        extract = p4_hlir.hlir.p4_parser.parse_call.extract
        default = p4_hlir.hlir.p4_parser.P4_DEFAULT

        #get the first state after parse_upvn
        state = h_mg.p4_parse_states['parse_upvn'].branch_to[default]
        start = h_mg.p4_parse_states['start']

        #reconnect the start state with the first state
        start.branch_to[default] = state
        state.prev.pop()
        state.prev.add(start)

        #remove the parse state from the graph
        h_mg.p4_parse_states.pop('parse_upvn')

        #extract the custom header at the beginning of the first state
        virtual = h_mg.p4_header_instances['upvn']
        state.call_sequence.insert(0, (extract, virtual))
```

4.3 Merging Headers and Metadata

To merge the headers of multiple programs, we compare each header of the program being added with the headers already present in the merged program. This comparison is performed in order to determine whether, each time, an equivalent header already exists in the merged program. As explained in Chapter 3, we have defined several types of equivalency between two headers, and each type of equivalence requires different considerations to ensure *correctness* is guaranteed by our merging process.

In order to determine what equivalence type exists between two headers, we apply the function shown in Listing 4.5.

This function first verifies if the number of fields present in both headers is the same (line 4). This verification is performed as a different number of fields automatically excludes the possibility of the headers being *Strongly* or *Simply* equivalent, and so the comparisons performed to establish those types of equivalence can be ignored. If the number of fields is the same, the function compares the names of the headers, followed by a comparison of the widths and names of its fields (lines 10 through 17). In case any of the comparisons fails, the variables declared in lines 2 and 3 are updated, changing their values to False, depending on whether the name (of either the header or any of its fields) or field width is different. After the fields are compared, the function verifies if all the fields have the same width, meaning the headers are either *Strongly* or *Simply* equivalent, with the distinction being made by verifying if the names are also the same. In case the field widths are not the same for each field, the function verifies if the total number of bits present in both headers is the same, and in case it is, the headers are considered *Weakly* equivalent.

If, however, the total number of fields is different, the portion of the function from lines 28 to 37 is executed. In this case, the function verifies only the total number of bits extracted by the headers (lines 30 through 34), and establishes that the headers are *Weakly* equivalent in case the verification is positive (line 37). If no equivalence is established between the two headers, the function returns `'No Equivalence'` as the result (line 38).

Listing 4.5: Function used to determine the equivalence type between two headers.

```
1  def check_equivalent_headers(header_instMG, header_instR):
2          sameWidthFields = True
3          sameNameFields = True
4          if len(header_instR.fields) == len(header_instMG.fields):
5                  if header_instMG.name != header_instR.name:
6                          sameNameFields = False
7                  mg_length = 0
8                  r_length = 0
9                  for i in xrange(len(header_instMG.fields)):
10                         mg_length += header_instMG.fields[i].width
11                         r_length += header_instR.fields[i].width
12                         if header_instMG.fields[i].width !=
13                                                 header_instR.fields[i].width:
14                                 sameWidthFields = False
15                         if header_instMG.fields[i].name !=
16                                                 header_instR.fields[i].name:
17                                 sameNameFields = False
18
19                 if sameWidthFields:
20                         if sameNameFields:
21                                 return 'Strong␣Equivalence'
22                         else:
23                                 return 'Simple␣Equivalence'
24                 elif mg_length == r_length:
25                         return 'Weak␣Equivalence'
26
27         else:
28                 mg_length = 0
29                 r_length = 0
30                 for i in xrange(len(header_instMG.fields)):
31                         mg_length += header_instMG.fields[i].width
32
33                 for i in xrange(len(header_instR.fields)):
34                         r_length += header_instR.fields[i].width
35
36                 if mg_length == r_length:
37                         return 'Weak␣Equivalence'
38         return 'No␣Equivalence'
```

Once the algorithm has established if there is or not an equivalent header in the merged program HLIR, the header is either shared, in the former case, or added separately, in the latter case. However, as mentioned previously, different equivalence types require different approaches when merging the various elements from the input programs, and so it is necessary to keep track of what type of equivalence was used to share a certain header, linking it with the header in the merged program HLIR that now represents it.

For this purpose, we have introduced three new dictionaries to the HLIR object, as seen in Listing 4.6. As the headers from the HLIR of the input program are being merged, they are placed in one of these three new dictionaries, depending on the type of equivalency that is established, and linked to the equivalent header in the merged HLIR (e.g., if header `header_added` from `hlir_-`

new is *strongly equivalent* to header header_merged from hlir_merged, our system maps the equivalence by declaring hlir_new.lStrongEq[header_new] = header_merged).

These dictionaries are then used, across the different merging modules, to quickly translate the header from the program being added to the header in the merged program, that represents it.

Listing 4.6: Dictionaries added to the HLIR object, used to identify equivalent headers.

```
hlir.lStrongEq = OrderedDict()
hlir.lSimpleEq = OrderedDict()
hlir.lWeakEq = OrderedDict()
```

4.4 Merging Parse States

To merge the parse states of an input program, we first look for equivalent states in the merged program HLIR. To do so, we look at the first header extracted by a state, and find the state in the merged program that extracts the equivalent header. In cases where such equivalent header is not found in the merged program, no equivalence can be established among the respective parse states (if there is no equivalent header in the merged program, there is also no equivalent state). Then, we verify if the remaining extracted headers (if any) also have equivalent headers being extracted in the state from the merged program, and if they are extracted in the same order. Then, we verify if the state operates on metadata that is used by other programs (we have modified the dumper.py and p4_table.py scripts from p4c to keep track of the metadata that is used by the programs). Finally, we verify if the states are in a compatible topological order.

4.4.1 Topological Order

As explained in the previous chapter, parse states are merged in topological order, from the lowest to the highest topological level. We have introduced this notion of topological levels in parse graphs, which is not present in the original HLIR object. Our mechanism assigns a topological level to each state, by recursively traversing the parse graph as shown in Listing 4.7.

Listing 4.7: Recursive function used to assign topological levels to parse states.

```
def fill_topo_order(h_mg, h_r):
        #attributing topological levels to the merged program
        recursive_fill(h_mg.p4_parse_states['parse_upvn'], 0)

        #attributing topological levels to the added program
        recursive_fill(h_r.p4_parse_states['start'], 0)

#function used to give each state a topological level
def recursive_fill(curr_state, level):
        states = []
        #get all states reachable by this state (no duplicates)
        #removing duplicates reduces the number of iterations
        for key , state in curr_state.branch_to.items():
                if state not in states:
                        states.append(state)

        #if the level received from the last state is greater
```

```
18        #than the one the state currently has, we update
19        if level > curr_state.topo_level:
20            curr_state.topo_level = level
21
22        #for every reachable state, call the function with
23        #the current state's topological level + 1
24        for state in states:
25            #if the next node is a table, then we have reached Ingress
26            if not isinstance(state, p4_hlir.hlir.p4_table):
27                recursive_fill(state, curr_state.topo_level + 1)
```

This mechanism receives the two HLIRs being merged (line 1), and assigns as a level to a state the length of the longest possible path in the graph to reach the state. This level is stored into the class attribute `topo_level` (line 20) of the state's object.

Since our algorithm merges the states present in the added HLIR as they are in the parse state dictionary, we must sort the states using the topological levels before the merge starts. The function used to sort the states can be seen in Listing 4.8.

Listing 4.8: Function used to sort the states by their topological level.

```
 1 #function used to sort h_r states by topological level,
 2 #ensuring states are added in the correct sequence
 3 def sort_parser_states(h_r, h_mg):
 4         #dict that contains the states of h_r, organized by topo.level
 5         #e.g., (0,[start]),(1,[ethernet]),(2,[ipv4]),(3,[tcp,udp])
 6         topo_level_dict = OrderedDict()
 7         for name, state in h_r.p4_parse_states.items():
 8                 level = state.topo_level
 9                 if level not in topo_level_dict:
10                         topo_level_dict[level]= [state]
11                 else:
12                         topo_level_dict[level].append(state)
13
14         #sorted dict containing all states
15         newDict = OrderedDict()
16         #for each topological level of the graph
17         for i in range(len(topo_level_dict)):
18                 #if only one state in this topo level
19                 if len(topo_level_dict[i]) == 1:
20                         #simply append to the dict
21                         newDict[topo_level_dict[i][0].name] =
22                                                 topo_level_dict[i][0]
23
24                 #if multiple states are at the same level, sort based on
25                 #the topo lovel of the equivalent state in h_mg
26                 else:
27                         sorted_topo_level =
28                           sort_same_topo_level(topo_level_dict[i],h_mg,h_r)
29                         for state in sorted_topo_level:
30                                 newDict[state.name] = state
31
32         h_r.p4_parse_states = newDict
```

As it can be seen in the listing above, the function *sort_parser_states* handles topological levels that contain a single state and those that contain multiple states differently. In fact, an extra sorting operation (line 27) is required to establish the order in which states in the same topological level must be merged. The importance of this mechanism was explained in the previous chapter, in

Section 3.5.2.

4.4.2 Sharing Parse States

Once an equivalence between two parse states is established, our system shares those states. The sharing mechanism, described in Chapter 3, is the same for both *Strongly* and *Simply* equivalent states. However, it changes when states are *Weakly* equivalent due to the fact that one of the shared states may select on fields that no longer exist in the merged program, as *Weakly* equivalent headers have a different structure.

Listing 4.9: Finding fields not present in the merged program

```
for fieldR in headerR.fields:
    offsetR = fieldR.offset
    widthR = fieldR.width
    found = False
    for fieldMG in headerMG.fields:
        if fieldMG.offset == offsetR and fieldMG.width == widthR:
            found = True
            break
    if not found:
        diff_fields.append(fieldR)
```

For the above reason, we track the fields that are not present in the merged program. This step is illustrated in Listing 4.9, where `headerR` represents the extracted header that will not be included in the merged program, whereas `headerMG` is the equivalent header already in the merged program.

Listing 4.10: Verifying if any translation must occur at this stage

```
#if there is no select statement, share state normally
if stateR.branch_on == []:
        share_state(h_r, h_mg, stateR, header_nameMG, h_meta,
                                first_merge, virtual)
#if there is select statement:
else:
        same = True
        #see if each field used is present in merged header
        for field in stateR.branch_on:
                if field in diff_fields:
                        same = False
        #if there is a direct translation from these fields to the ones
        #in the merged header, normal share
        if same:
                share_state(h_r, h_mg, stateR, header_nameMG,
                        h_meta, first_merge, virtual)

        #edge case: must translate from header_R fields to merged
        #header fields to select correctly
        else:
                fields_to_add = get_fields_to_add(stateR, headerMG)
                modify_mg_transitions(state_MG, fields_to_add)
                modify_r_transitions(stateR, state_MG, headerR,
                                first_merge, virtual)
```

Once a weak equivalence between two states has been established, three possible cases can occur (the related code is reported in Listing 4.10): (1) the state being added does not select on

Listing 4.11: Finding the set of equivalent fields to be added

```
def get_fields_to_add(parser_stateR, headerMG):
        fields_to_add = []
        for fieldR in parser_stateR.branch_on:
                for fieldMG in headerMG.fields:
                        if fieldMG.offset > fieldR.offset +
                                                fieldR.width -1
                            or fieldMG.offset + fieldMG.width -1 <
                                                fieldR.offset:
                                pass
                        else:
                                fields_to_add.append(fieldMG)
        return fields_to_add
```

any field, and so the sharing can occur normally, using the standard state sharing mechanism (line 2), (2) the state selects on a set of fields that exist in the merged program, meaning it can also be shared using the standard mechanism (line 14), or (3) the state selects on a field that does not exist in the merged program, and translation is required (line 20).

For the last case, the fields selected on the state being added that no longer exist (because the equivalent header in the merged program has a different structure) must be represented in the shared state's select statement, by including the intersection of fields from the equivalent header that contain all the bits present in the missing field. The steps performed to determine the set of fields that must be added are reported in Listing 4.11.

After the fields have been added to the shared state, and the pre-existing transitions modified to ignore them, the transitions from the state being added are also modified, to conform to the set of fields being selected on the shared state. More precisely, the transitions must be re-written to consider the fields that are selected (and their order) in the shared state, ignoring the fields that were not originally present in the added state. Once these modifications are applied, the transitions can be placed in the shared state.

4.4.3 Backtracking transitions

In our system, parse states are integrated into the merged program one by one, by either being shared or simply added. However, parse graphs are represented by multiple objects within a program's HLIR, with objects possibly linked to other objects. So, when merging a parse state into a different HLIR, we must modify the original links in these objects accordingly to the structures and objects already presented in the merged program.

To better depict this problem, we consider the scenario illustrated in Figure 4.2 as an example: Two programs to be merged, here named as A and B, both containing two states, namely {`ethernetA`, `ipv4A`} and {`ethernetB, ipv4B`}. We assume that equivalences exist between respective ethernet and IPv4 states of both programs, and that program A serves as the base program (i.e., states from B are added to A). When state `ethernetB` is shared with `ethernetA`, we copy its transitions to the merged state's transition dictionary. However, one transition of the state `ethernetB` references the state `ipv4B` in B. If we included this transition in the merged program and then subsequently shared IPv4 states, thus only keeping the `ipv4A` state in the merged program, we would achieve an incorrect merged result with the ethernet state containing two transitions - one to ipv4A, present in the merged program, and one to ipv4B, not present in the merged program. However, it is also not possible for us to add a transition to the ipv4 state in A when merging ethernet states, since at that stage we do not yet know if the IPv4 states will be shared.

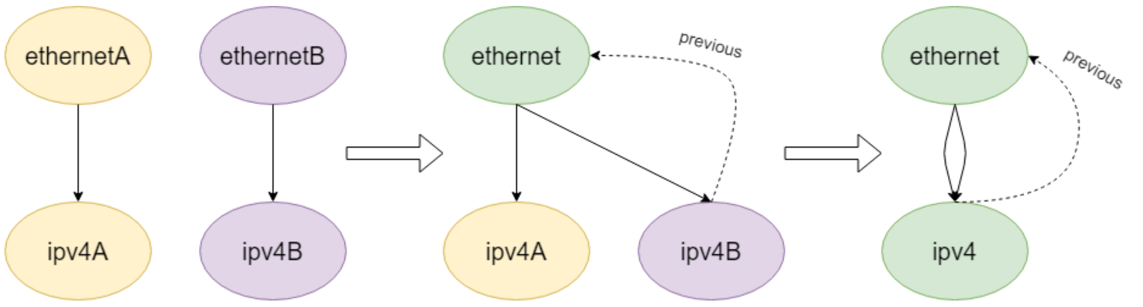


Figure 4.2: Step-by-step illustration of the merging process for the two example programs. The two graphs are temporarily linked until all the states have been merged.

Listing 4.12: Changing the reference to the previous state in the HLIR being added

```
1 #add transitions
2 for key, state in parser_stateR.branch_to.items():
3         h_mg.p4_parse_states[state_MG.name].branch_to[key] = state
4         #change the reference to the previous state
5         if not isinstance(state, p4_hlir.hlir.p4_table):
6                 state.prev.add(state_MG)
```

To overcome the above problem, our solution allows transitions of states in the merged program to be modified at a later stage to reflect those states which are shared later in the merging process.

We achieve this in two steps. In the first step, in the related code reported in Listing 4.12, we perform two operations at the end of the state sharing mechanism: (1) adding the original transition from the state that was shared (line 3), with those transitions still referencing objects in the HLIR being added (considering our previous example, adding the transition to `ipv4B` - here called `state` - to `ethernetA` - here called `state_MG`), and (2) modifying the next state in the added HLIR to contain a reference to the new state in A (line 6), instead of the old state in B (considering again our previous example, modifying `ipv4B`, by saying it is now reached through `ethernetA`, instead of `ethernetB` - here called `parser_stateR`).

Listing 4.13: Backtracking the old transitions, changing the state it transitions to if it has been shared

```
#Change parse state object in previous transitions
for state in parser_stateR.prev:
        for key, tempState in state.branch_to.items():
                if tempState == parser_stateR:
                        state.branch_to[key] = state_MG
```

In the second step, reported in Listing 4.13, once we have shared a state, and so we will not include it in the merged program, every transition to this state must be modified, referencing the shared state instead. To achieve this, we visit and verify every state that precedes the shared state to modify the transitions to it. In case a transition is referencing the state (line 4), it is replaced with the correct state (line 5), e.g., in the example above, when sharing `ipv4B` with `ipv4A`, we must go to the state that precedes it (that is now `ethernetA`), and upon identifying the transition that was added in the previous step, modifying the reachable state, from `ipv4B` to `ipv4A`.

4.5 Optimizing Transitions

In order to reduce the amount of resources required to store the merged program, our system identifies in, and removes duplicate transitions from, the merged parse graph. As explained in Chapter 3, we consider a set of transitions in a state *S* to be redundant if (a) the number of transitions with the same transition value is equal to the number of programs sharing the state, and if (b) these transitions have the same destination state.

As a first step towards this optimization, we determine how many input P4 programs are sharing a state, achieved with the function reported in Listing 4.14. The function, named *get_shared_-program_count*, determines the total number of programs sharing a certain state by counting the number of unique program IDs present in the select entries dictionary. To achieve this goal, the function begins by translating the select entries to binary format (line 5). This is necessary as the program's ID must be extracted from the value being used as key (line 15). Once the ID has been separated from the original entry value, it is added to a list of unique IDs (line 17), that is then returned. The number of programs sharing this state is equal to the size of the returned list.

Listing 4.14: Function used to get the total number of programs sharing a certain state.

```
1  def get_shared_program_count(state):
2          temp_list = []
3          for key, nextState in state.branch_to.items():
4                  #original key (i.e., pvid + field in binary format)
5                  originalBin = format(key[0], 'b')
6                  width = 0
7                  for field in state.branch_on:
8                          if isinstance(field ,tuple):
9                                  width = width + field[1] - field[0]
10                         else:
11                                 width = width + field.width
12                 #original key with leftmost bits added if needed
13                 originalBin = originalBin.zfill(width)
14                 #program id, present in the first 4 bits
15                 program_id = originalBin[:4]
16                 if program_id not in temp_list:
17                         temp_list.append(program_id)
18         return len(temp_list)
```

Once the programs count is computed, we iterate through the select entries of the state, as shown in Listing 4.15. We add the transitions that are equivalent to a list, named `temp_remove`, in case both the value and the mask of the two entries are the same (line 30). Those added transitions cannot yet be removed, since we must first ensure that every program sharing this state contains this transition.

Listing 4.15: Finding candidate transitions for deletion

```
#list of redundant cases to be removed
temp_remove = []
#for every transition of the state
for key, next_state in state.branch_to.items():
        if key not in tempDict:
                continue
        keyBin = format(key[0], 'b')
        keyBin = keyBin.zfill(width)
        #value of the select entry without the program ID
        realValue = keyBin[4:]
        mask = format(key[1], 'b')
        mask = mask.zfill(width)
        #mask of the select entry without the program ID
        realMask = mask[4:]
        #look for equivalent transitions
        for key2, next_state2 in state.branch_to.items():
                if key == key2:
                        continue
                keyBin2 = format(key2[0], 'b')
                keyBin2 = keyBin2.zfill(width)
                realValue2 = keyBin2[4:]
                mask2 = format(key2[1], 'b')
                mask2 = mask2.zfill(width)
                realMask2 = mask2[4:]
                #if the two transitions have the same transition condition
                #and following state then they are temporarily added to
                #the remove list
                if realValue == realValue2 and realMask == realMask2
                            and next_state == next_state2:
                        temp_remove.append(key2)
```

Listing 4.16: Deleting all transitions that are equivalent to the current transition

```
 1  #if every program sharing this state has the same transition as
 2  #the current (key, next_state) pair remove all redundant transition
 3  if len(temp_remove) == prog_count - 1:
 4          for case in temp_remove:
 5                  tempDict.pop(case)
 6          #change the remaining transition, from id+field mask f+f
 7          # to 0+field mask 0+f
 8          if realValue == '':
 9                  realValue = '0'
10          if realMask == '':
11                  realMask = '0'
12
13          #if the select entry is empty, then it was originally a default
14          if (int(realValue,2),int(realMask,2)) == (0,0):
15                  default = p4_hlir.hlir.p4_parser.P4_DEFAULT
16                  tempDict[default] = tempDict.pop(key)
17          else:
18                  tempDict[(int(realValue,2),int(realMask,2))]=
19                                          tempDict.pop(key)
```

As shown in Listing 4.16, if all the other programs sharing the state contain an equivalent transition (line 3), those transitions can be deleted from the merged program. This is achieved by removing the transitions placed in the temporary list from the select entries dictionary (line 5). Finally, the corresponding remaining transition is replaced by a transition with the same value, but without the program ID (lines 14 through 19).

4.6 Summary

In this chapter, we presented the implementation of P4Visor++, our system to merge parser graphs from multiple P4 programs. We presented the system's structure in Section 4.1. We described the integration of our custom program elements in the merge process in Section 4.2. In Section 4.3 and in Section 4.4, we explained respectively how equivalent headers and equivalent parse states are merged. To conclude, in Section 4.5, we described how we eliminate redundant transitions in the parse graph of the merged program so to reduce the resources required for storing it.

Chapter 5

Evaluation

To evaluate the ability of our solution to efficiently and correctly merge parse graphs from different P4 programs, we performed two different types of experiments. The first set, described in detail in Section 5.2, aims to demonstrate that our merging algorithm preserves *correctness*, i.e., the paths in the parse graph of any input program are preserved in the merged program, and guarantees *isolation*, that is, only states present in the parse graph of a certain input program will be reached by packets destined for that input program in the merged program. The second set of tests, described in detail in Section 5.3, aims to evaluate the efficiency of our solution, by comparing the size and the complexity of a merged program generated by our merging algorithm with the results achieved by the state-of-the-art system P4Visor. By comparing the total number of states and transitions in the parse graphs of the input programs and of the merged program, we aim to showcase the reduction of memory space required to physically store a parse graph on a P4-programmable target enabled by our merging algorithm.

The rest of the chapter is organized as follows. In Section 5.1 we describe the environment used for our tests and the set of P4 programs selected for our experiments. Section 5.2 and Section 5.3 describe, respectively, the two different sets of experiments we have performed, and discuss the main results.

5.1 Testing environment

In this section, we present all the components used for testing and evaluating our algorithm for merging parse graphs, namely the P4 programs and the software tools. Similarly to the implementation, all the evaluation was performed on a COTS computer, equipped with a 3.4 GHz AMD Ryzen 5 2600 CPU and 8GB of memory, running Ubuntu 18.04.

5.1.1 P4 programs for testing

We selected ten different programs, independently written by different parties and available on the web, to create a testing set for our evaluation. We have chosen a balanced combination of realistic programs whose parse graphs show different degrees of similarity between themselves.

Through this set we aim to highlight the main challenges of the merge process and the benefits of our merging solution. In Table 5.1, we provide a short description of each of the ten programs in our set.

ID	Names	# of States	Protocols	# of Edges	Description of the Parse Graph
P1	**Simple Router [17]**	3	Standard: 2 (Eth, IPv4) Custom: 0	4	Basic parser extracting standard protocols through two states.
P2	**Flowlet [18]**	4	Standard: 3 (Eth, IPv4, TCP) Custom: 0	6	Extends the previous graph by extracting one more standard protocol.
P3	**Heavy Hitter [19]**	4	Standard: 3 (Eth, IPv4, TCP) Custom: 0	6	This parse graph is an exact copy of P2's one, used to demonstrate perfect merge with different programs that share the same parse graph.
P4	**Port Knock [20]**	4	Standard: 3 (Eth, IPv4, TCP) Custom: 0	6	Same parse graph as in P2, but the field organization of the TCP header is different, to illustrate the merge of weakly equivalent headers.
P5	**MC_nat [21]**	4	Standard: 3 (Eth, IPv4, UDP) Custom: 0	6	Simple IPv4 to UDP parse graph, used to highlight cases where only portions of the graph can be shared.
P6	**Timestamp [22]**	5	Standard: 4 (Eth, IPv4, UDP, RTP) Custom: 0	7	Slightly more complex version of P5's parse graph, with an RTP state at the end.
P7	**ECMP [23]**	5	Standard: 4 (Eth, IPv4, TCP , UDP) Custom: 0	8	Parse graph that can transition to both TCP and UDP states, with the particularity that its "parse_ipv4" state selects on an extra field.
P8	**mTag edge [24]**	5	Standard: 3 (Eth, Vlan, IPv4) Custom: 1	10	The parse graph includes VLAN and mTag headers between ethernet and IPv4.
P9	**Source Routing [25]**	3	Standard: 0 Custom: 2	5	Parse graph expects two custom headers in the beginning of the packet. So, there is a select statement in the "start" state which is empty in all the graphs above, since those rather transition directly to a state to extract the ethernet header.
P10	**Simple Router with ARP [26]**	5	Standard: 3 (Eth, ARP, IPv4) Custom: 1	8	Similarly to Source Routing, there is a select statement in the "start" state, that transitions to either "parse_ethernet" or "parse_CPU". The latter then transitions to "parse_ethernet", followed by either an IPv4 or ARP header.

Table 5.1: Our testing set is built from ten programs retrieved from independent on-line sources. For the sake of easy reference later in this chapter, we summarize here in this table some of the characteristics of their parse graphs. The complete parse graphs of these programs are illustrated in Appendix A.

Because our solution merges only the parse graphs of the programs, all of the other program's logic is replaced by a simple custom input-to-output port forwarding mechanism, for every program in our set. Additionally, parsers have been modified to parse, set and extract our custom header, whose fields are also used for testing (this will be explained in detail in Section 5.2).

5.1.2 Software Tools

For testing the execution of the P4 programs used in our experiments, we have used the reference P4 software switch [15], also called *bmv2* (which stands for behavioral model 2). The *bmv2* software switch is a tool for developing, testing and debugging data planes written in P4, developed and maintained as an open-source project by the P4.org community. *Bmv2* can load an intermediate representation of a P4 program, generated by a compiler, and implement the packet-processing behavior specified in that program. We have used the *p4c* reference P4 compiler [14] to compile our testing programs. The *p4c* compiler can produce intermediate representations of P4 programs for bmv2 through a target-specific back-end compiler. We have also used another back-end available with *p4c*, namely *p4-graphs*, to produce most of the graphs reported in this manuscript.
To run our tests, both the original and the merged programs were compiled and deployed on the bmv2 virtual switch. At start-up time, the bmv2 switch's ports can be connected to virtual interfaces on the host machine. The interfaces connected to the switch can be used to inject traffic into the switch and to sniff the traffic produced in output to verify that the packet processing behavior has been executed accordingly to the loaded P4 program.
To create custom packets to be sent through the switch, as well as to sniff and analyze the forwarded packets, we developed three applications based on the python-based Scapy library [27]. Scapy allows a developer to specify the protocol stack (including custom-made protocols) and payload in network packets, inject the crafted packets in the network, as well as receive and inspect packets.

The experimental setup consists of one sender and two different receivers connected by the virtual switch, as shown in Figure 5.1. The sender sends custom traffic to the virtual switch *bmv2*. This traffic is generated according to an auxiliary file which we have compiled describing the protocol headers expected in packets for a certain program. The two receivers are named `receiver_shadow.py` and `receiver.py`. The former is used for tracking the parser states visited in the original programs, while the latter is used for comparing the parsing results produced by the standalone execution of the original programs against the results produced by the merged program. This latter step is performed to assess whether the *correctness* and *isolation* properties have been preserved.

5.2 Verifying Isolation and Correctness

To achieve our goal of verifying the *correctness* and *isolation* of merged P4 programs, we have deployed a mechanism that allows us to precisely track the parser states visited by each packet as it is processed by our program in the switch. This mechanism consists in:

1. having a custom header (*upvn*) to carry the program ID (*pvid*) and a bitmap (*p_map*) of the traversed parser states within packets;

2. attributing an ID (0-15) to each parser state;

3. in each parser state, setting the state ID into the bitmap and storing the updated bitmap into a metadata variable;

4. in the ingress pipeline, copying the metadata field written during the parsing stage into the custom header *p_map* field before forwarding the packet to an output port.

We first apply the above mechanism to both the original programs and the merged ones. Afterwards, by comparing the *p_map* values obtained with the original programs against the *p_map* value obtained with the merged program, we can assess whether or not a certain packet has traversed only the correct states in the original program. As it can be seen in Figure 5.1, a receiver, e.g., *host2*, of the packets processed by our modified programs can inspect the custom *upvn* header to know the states visited by the packets through the parser stage.

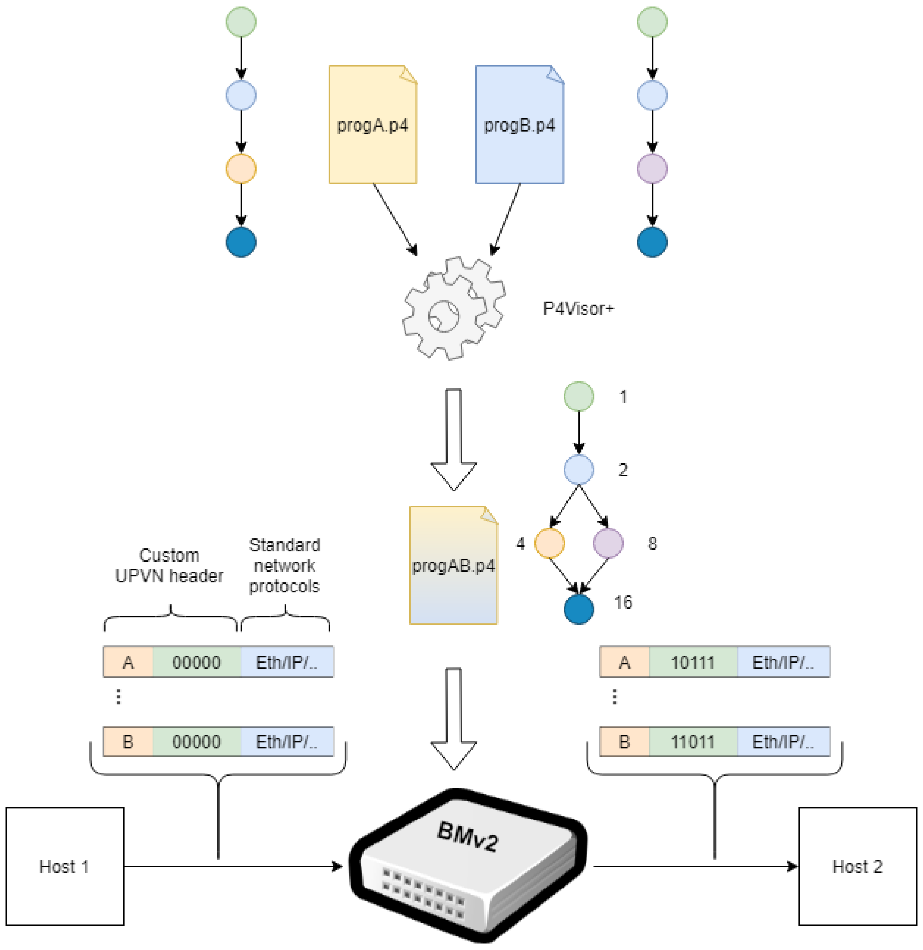
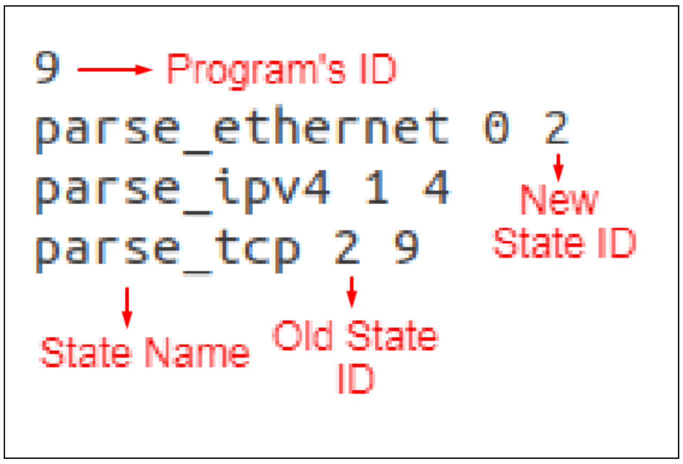


Figure 5.1: Illustration of the mechanism used to test *isolation* and *correctness*. In the example, packets processed by the virtualized `progA.p4` must visit the states with IDs {1,2,4,16}. After being processed by `progA.p4`, the packet has a `p_map` field value of `10111`, meaning it visited all states except the one with an ID of 8 (fourth position from the right, or 2^3), demonstrating it is *correct* and *isolated*.

Leveraging the above mechanism, this test consists of the following steps:



```
9
parse_ethernet 0 2
parse_ipv4 1 4
parse_tcp 2 9
```

Figure 5.2: Translation file for the flowlet program (P2) produced at point 1: The first line of the program contains the ID of the program, then the next lines list parser states where original and modified state IDs are noted.

1. We merge any combination of programs using our P4Visor++: This step generates i) a translation file for each input program, which matches by IDs the parse states in the original program to the states in the merged program (see Figure 5.2 for an example) and ii) a compiled version of the merged program.

2. Using the original IDs in the translation file created in step (1), we manually create a copy of the program to be tested (this process needs to be done only once for each program), which:

   - parses and extracts our custom upvn header, updates the p_map bitmap (see Figure 5.3 for an example),
   - replaces the original processing logic with our custom simple input-to-output port forwarding logic.

3. We compile the program created in step (2) with the *p4c* compiler and we run it on the bmv2 switch.

4. We create an auxiliary text file `<program_name>_packets.txt` that contains the definition of the testing packets.

5. We run simultaneously `receiver_shadow.py` and `sender.py` providing them with the program's name as an argument. This step will generate a file (named `<program_name>_original_result.txt`) with the result of sending the testing packets specified in the auxiliary file produced at step (4) through the program created in step (2) and executed in step (3).

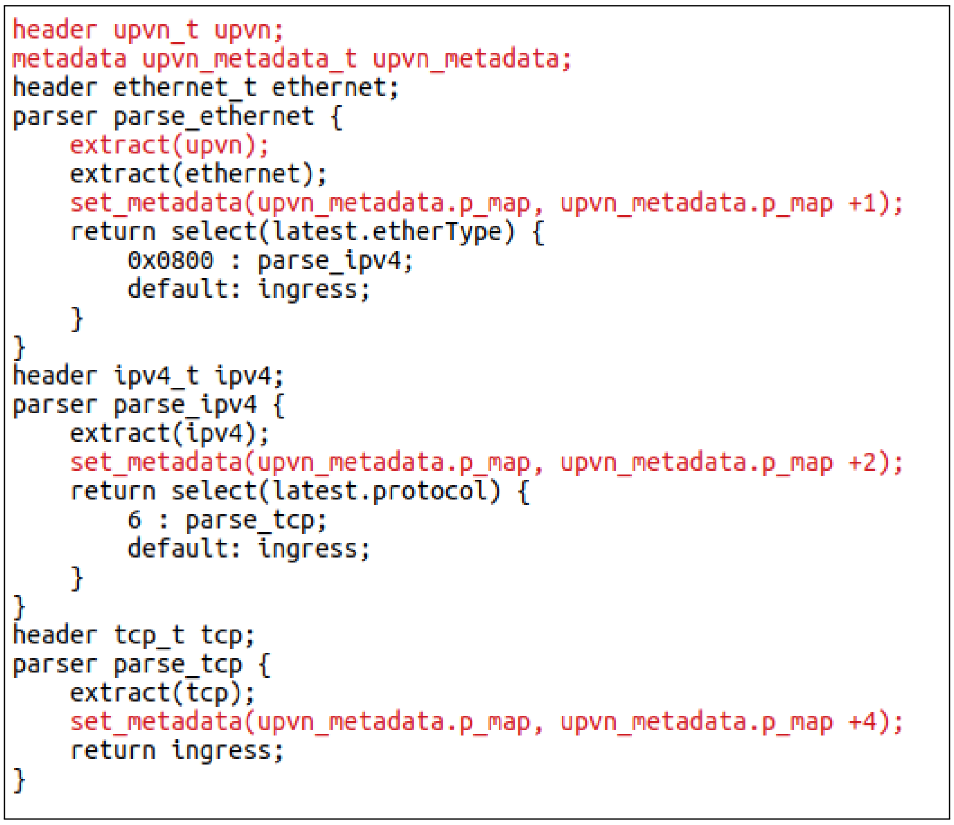
6. We shut down the switch and restart it with the merged program produced at step (1).

7. We run `receiver.py` and `sender.py`, specifying the name of the program to be tested.

8. We compare the visited states in the merged program with the visited states in the original program, using the translation file generated in step (1).

9. We output a positive result if and only if the states visited in the merged program are exactly the ones visited in the original program.

We better illustrate the test and its steps through an example with the flowlet (P2) program by merging all the programs of Table 5.1 with our P4Visor++. The translation file for P2 in our example is reported in Figure 5.2.



```
header upvn_t upvn;
metadata upvn_metadata_t upvn_metadata;
header ethernet_t ethernet;
parser parse_ethernet {
    extract(upvn);
    extract(ethernet);
    set_metadata(upvn_metadata.p_map, upvn_metadata.p_map +1);
    return select(latest.etherType) {
        0x0800 : parse_ipv4;
        default: ingress;
    }
}
header ipv4_t ipv4;
parser parse_ipv4 {
    extract(ipv4);
    set_metadata(upvn_metadata.p_map, upvn_metadata.p_map +2);
    return select(latest.protocol) {
        6 : parse_tcp;
        default: ingress;
    }
}
header tcp_t tcp;
parser parse_tcp {
    extract(tcp);
    set_metadata(upvn_metadata.p_map, upvn_metadata.p_map +4);
    return ingress;
}
```

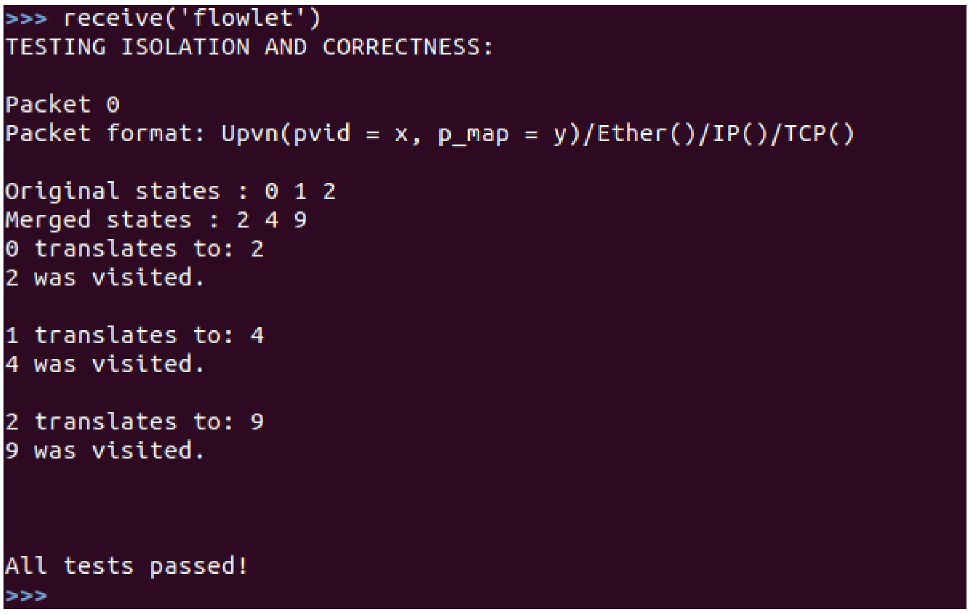
Figure 5.3: Copy of the original flowlet program with the modified parse states. In red, the `set_metadata` statements updating the `p_map` field with the corresponding value for each state.

By using the states' information contained in the translation file, we instrument a copy of the flowlet program with our custom parser operations as described in step (2), which is shown in Figure 5.3. Once the program copy is compiled and loaded on the bmv2 switch (3), we run our `sender.py` and `receiver_shadow.py` scripts (5), which send and receive, respectively, the testing packets defined in the `flowlet_packets.txt` file, which has been previously manually prepared (4). In this example, that file only contains one testing packet in the format :

```
Upvn(pvid = x, p_map = y)/Ether()/IP()/TCP()
```

The result is the `flowlet_original.txt` file listing on a single line the IDs sequence of the parser states visited by the testing packet.

Afterwards, we load the bmv2 switch with the merged program (6) and start the `sender.py`



```
>>> receive('flowlet')
TESTING ISOLATION AND CORRECTNESS:

Packet 0
Packet format: Upvn(pvid = x, p_map = y)/Ether()/IP()/TCP()

Original states : 0 1 2
Merged states : 2 4 9
0 translates to: 2
2 was visited.

1 translates to: 4
4 was visited.

2 translates to: 9
9 was visited.



All tests passed!
>>>
```

Figure 5.4: Output of `receiver.py` for the flowlet program example: The line '`Merged states : 2 5 8`' corresponds to the states visited by by the testing packet in the merged program, whereas the line '`Original states : 0 1 2`' is read from `flowlet_original.txt`.

and `receiver.py` scripts (7). By specifying the program name to be tested as an argument to the sender script, the sender injects the custom packet into the switch and the receiver stores its content on arrival. The `receiver.py` script determines whether the parse graph in the merged program preserves correctness and isolation for the original program tested. It does so by comparing the content of the custom header in the received packet (this packet is defined in `flowlet_packets.txt`) and the values stored in both the translation file and the `flowlet_original.txt` file. The result of this for the flowlet (P2) example is illustrated in Figure 5.4.

The above test was performed individually for every input program over several combinations of the testing programs we have merged with P4Visor++. These combinations included simple merge cases (that is, program pairs) as well as the more complex cases presented in Section 5.3. In all our tests, our mechanism confirmed that only the expected states were visited by the packets for the input programs, regardless of the number and combination of the merged programs, demonstrating that the *correctness* and *isolation* properties are guaranteed by our merging algorithm.

5.3 Evaluating Parse Graph Complexity

To measure the complexity of the merged programs produced by P4Visor++, we consider three different metrics in the parse graph of the merged program - number of headers, number of states, and number of transitions - and compare those metrics against the results obtained with the state-of-the-art P4Visor system. The choice of these metrics has been driven by real physical limitations on target switching hardware. More precisely, target hardware features limited amount of memory to store headers, metadata structures and parser graphs. As the number of programs being merged increases, it is important that a virtualization solution considers these possible target hardware limitations. Since data regarding the usage of such resources is not made available in the P4Visor work, and the implemented version does not support multiple programs, we obtain the values corresponding to P4Visor by simulating its parsing mechanism: performing a naive merge of the parse graphs, that is, assuming states are not shared and simply added to the merged graph.

5.3.1 Comparing Merge Efficiency Against P4Visor

We have performed several experiments using the programs in our set (please refer to Section 5.1.1), gradually decreasing the degree of similarity between the parse graphs to merge. Test A consists in merging programs $\{P2, P3, P4\}$. All these programs feature a simple parse graph extracting the headers Ethernet, IPv4 and TCP. The goal of this experiment was to compare the results of merging programs with a high degree of similarity (as the parse states are identical). Test B consists in merging programs $\{P1, P5, P6, P7\}$. With this experiment, we aim to showcase how our solution performs when merging programs that display different parse graphs, where only part of the resources can be shared (all these programs extract Ethernet and IPv4, only some extract UDP). The last experiment, Test C, is composed of programs $\{P8, P9, P10\}$. This set contains programs with a low degree of similarity, as the parse graphs are very distinctive. With this experiment, we intend to compare the results achieved by our solution when merging graphs that have few resources that can be shared. Overall, as illustrated in Table 5.2, our merging solution

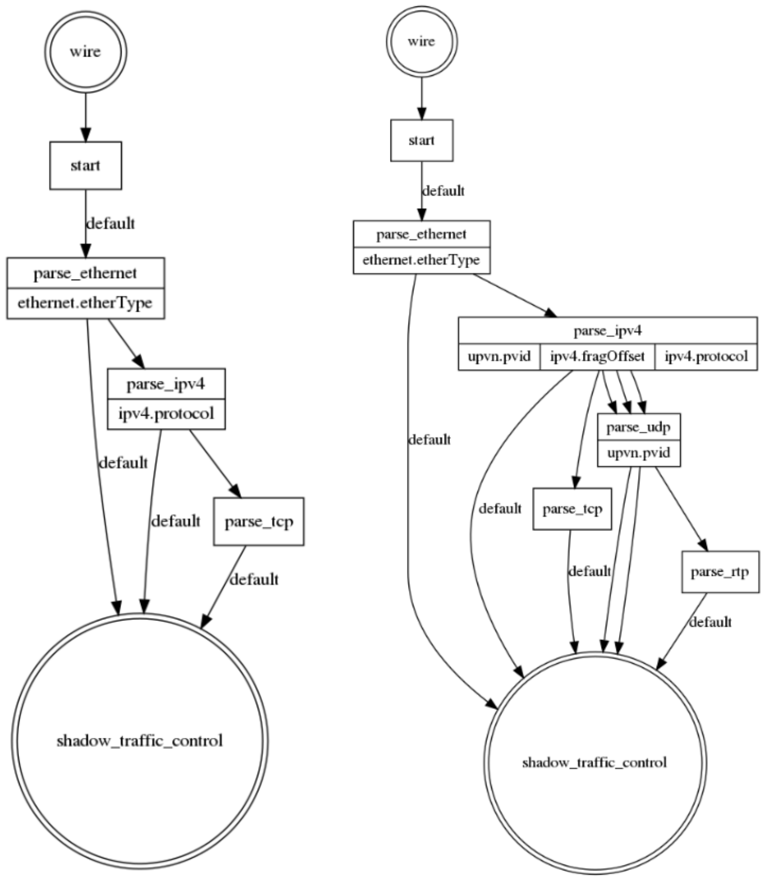
	Transitions			**Headers**			**States**		
	P4Visor	P4Visor++	Gain	P4Visor	P4Visor++	Gain	P4Visor	P4Visor++	Gain
Test A	19	6	68%	17	10	41%	11	4	64%
Test B	26	13	50%	21	11	48%	15	6	60%
Test C	24	21	13%	16	11	31%	12	9	25%

Table 5.2: Results with Merging Multiple Programs

always reduces the amount of resources used to represent the parse graph when compared with P4Visor, showing savings of up to 68% in transitions, 48% in headers and 64% in states.

In Test A, our solution achieves perfect merging results with regards to transitions and states, meaning the merged parse graph is identical to the original graphs. This is better illustrated by Figure 5.5a, showing the original graphs alongside the merged graph. Our solution finds equivalences between all the extracted headers and corresponding parse states among the three programs. Since the values used to transition between states are also the same, the three graphs can be represented using only the resources of a single graph. Additionally, our custom parse state is removed (as every program transitions to the same state from `start`) and our custom header is extracted in the first parse state (i.e. `parse_ethernet`).

In Test B, we achieve similar results comparatively with Test A with regards to headers and states, but suffer a considerable reduction in the efficiency gain obtained over P4Visor regarding



(a) (b)

Figure 5.5: Resulting graphs of the first two tests. Respectively, Test A is represented by (a), Test B is represented by (b). The input graphs for each test can be seen in Appendix A.1 and A.2, respectively.

transitions. As can be seen in Figure 5.5b, this decrease in efficiency is justified by the presence of replicated transitions from states `parse_ipv4` to `parse_udp`, and from `parse_udp` to the *Ingress* stage (here represented by the table `shadow_traffic_control`). These transitions must be replicated since a set of equivalent transitions (that is, transitions that originate from the same state, contain the same original select entry and transition to the same state) can only be reduced to one if every program sharing the state contains such transition. In this case, only three of the four input graphs contain a transition from `parse_ipv4` to `parse_udp` (program P1 does not contain a UDP parse state), forcing the corresponding transitions in the merged graph to include the program ID. A similar condition occurs in the state `parse_udp`, where program $P6$ does not contain a default transition to the *Ingress* stage.

Finally, in Test C, our solution shows only a minor decrease with resources used compared with P4Visor. This occurs because the three graphs are very different, and they do not have many equivalences which can be leveraged to share resources in the merged parse graph. In fact, our solution only manages to share three states among the three parse graphs, namely `ethernet`, `ipv4` and `parse_head`, as it can be seen in Figure 5.6. Besides, the last state in the merged

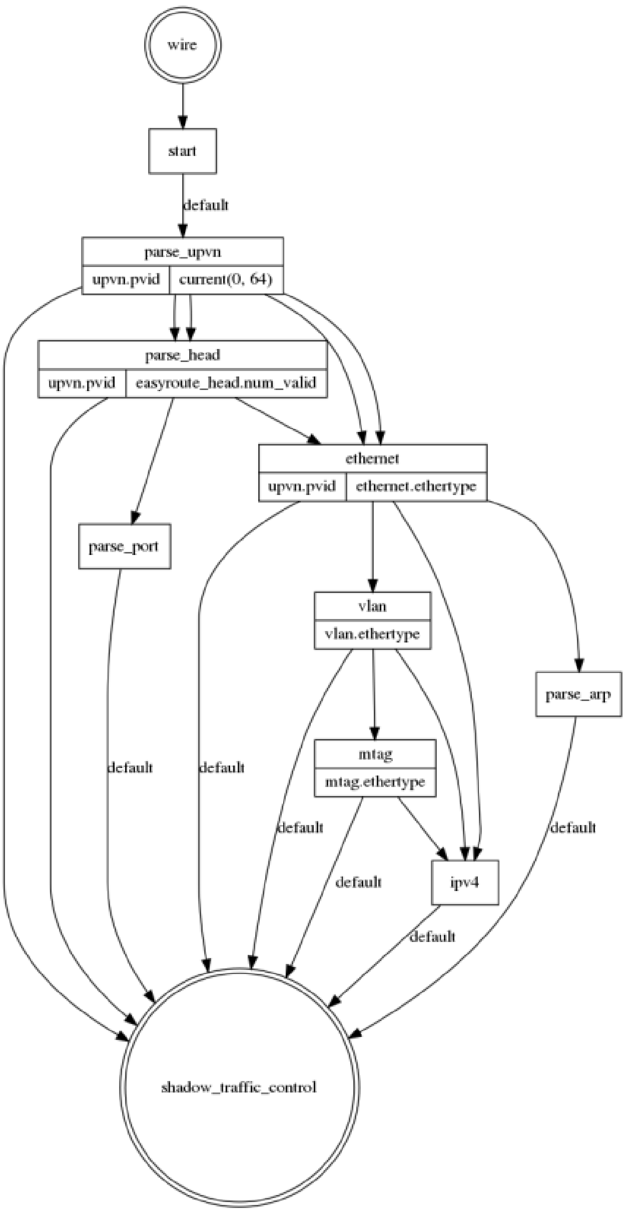


Figure 5.6: Resulting graph for Test C. The input graphs can be seen in Appendix A.3.

graph, `parse_head`, belonging to program $P9$, is shared with state `parse_cpu` from program $P10$, due to the fact that the headers extracted by these states are *weakly equivalent*. This special case highlights P4Visor++'s ability to improve the efficiency of the merging process by leveraging weakly equivalent headers, which may sometimes correspond to different protocols in the original input parse graphs.

5.3.2 Limitations

Despite the reduction in resource usage showcased in the previous section, there are specific conditions which may prevent our algorithm to produce optimal merge results. Throughout our study, we have found two specific conditions across the input graphs which can severely affect the results of our merging algorithm. In this section, we document those conditions alongside with simple strategies which can be used to overcome them.

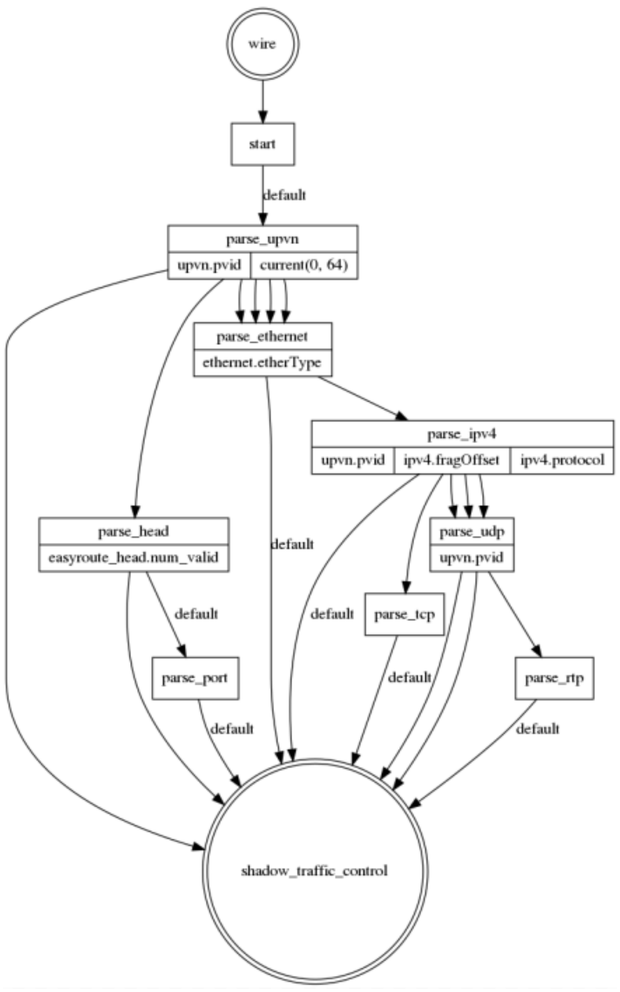


Figure 5.7: Parse graph generated by merging $P9$ with the programs in Test B

The first kind of condition refers to transitions in a shared state can only be shared if every program sharing that state contains the same transition. In our set of programs, an occurrence of this condition causes an increase in the number of transitions proportional to the number of input programs. We illustrate this with an example where $P9$ is merged with the programs from Test B $\{P1, P5, P6, P7\}$, shown in Figure 5.7. Every program in our set contains a default transition to `parse_ethernet` from `start`, except for $P9$, that does not extract Ethernet headers, and consequently, does not have that transition. Instead, $P9$ transitions to either another state or to the `Ingress` stage based on the first 64 bits in the current packet. For this reason, the default transition from `start` to `parse_ethernet` present in all the other programs cannot be shared. In Figure 5.7, we can see such transitions to the Ethernet parse state, one for each of the programs from Test B, originating from the state `parse_upvn`, in the merged graph.

The second kind of condition refers to detecting weak equivalences between states that do not extract the same protocols. This condition may also lead to an increase in the number of transitions for our algorithm. This problem can occur in our program set when we merge, for example, program $P10$ with program $P6$, due to the fact that our algorithm detects a *weak equivalence* between the `cpu_header` in $P10$ with the RTP header in $P6$. This equivalence is established because, despite the two headers representing different network layers, they have the same total width. When $P10$ is merged after $P6$, an equivalence is established between those two headers and the respective parse states are consequently shared. Afterwards, an equivalence is established between the states parsing Ethernet and IPv4, which are present in both $P10$ and the merged program. However, those states cannot be shared since Ethernet and IPv4 are now at a higher topological level than `parse_cpu` in $P10$, and at a lower level than RTP in $P6$. For this reason, our merging algorithm adds the Ethernet and IPv4 states from $P10$ to the merged program without sharing those. As a consequence, an extra transition from the `parse_upvn` state is added to reach the new Ethernet state, followed by a transition to the new IPv4 state. The resulting graph can be seen in Figure 5.8, where we can see `parse_rtp` replacing $P10$'s `parse_cpu_header`, and the new states being included.

If, however, we merge the programs in a different order, merging $P6$ after $P10$, then the resulting merged graph will contain fewer transitions (12 instead of 16) than the ones achieved with the previous programs sequence, as shown in Figure 5.9. This time, as we add program $P6$ first, our algorithm finds an equivalence between the states parsing Ethernet and IPv4 headers in the program being added and the respective states in $P10$, thus sharing those states. After such merge has been performed, our algorithm can no longer share the RTP state with the `cpu_header` state, despite the *weak equivalence* between the two still being detected.

It is important to notice that the merging result is also correct in the first case, as the `cpu_-header` header is renamed across the whole program, and $P10$ uses the extracted header in `parse_rtp`, as it contains the same number of bits. Yet, this example highlights that the order in which programs are merged may have an impact on the efficiency achieved by our merging algorithm.

5.3.3 Optimizations

As showcased in Subsection 5.3.2, the order in which input programs are processed by P4Visor++ may lead to non-optimal merged graphs because of the detection and merge of weakly equivalent headers. Hence, we have devised an additional mechanism for P4Visor++ to identify input programs sequences which lead to better merging results in average.

This mechanisms consists of a pre-processing stage where we look for program sequences in the input set that produces smaller graphs. We do so by merging the different permutations of the programs and by returning the sequence that produces the lowest amount of resources used. This is, however, a task that can not be quickly performed, as, for example, with ten programs, there exist 10! (3,628,800) different permutations. To reduce the computation complexity of this pre-processing stage, we build upon the observation that some input programs in the same set often have similar graphs. By considering that, we can often reduce the total number of programs

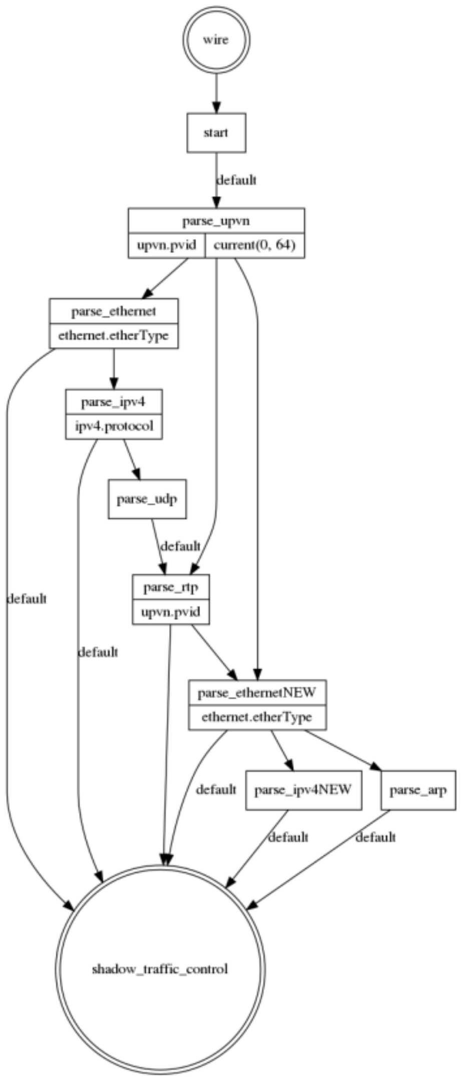


Figure 5.8: Parse graph resulting from the merge of $P6$ followed by $P10$

being used by the pre-processing stage by aggregating programs with the same parse graphs (e.g., programs $P2$, $P3$ and $P4$ have very similar graphs), and then only re-introduce them back in the input sequence once the optimal merge sequence has been determined, by placing them adjacent to the program that has the similar graph.

With the above aggregation heuristic that we have devised, we can compute the best sequence we observed for our programs set in a few minutes, instead of several days. The resulting parse graph is the one reported in Figure 5.10, containing 40 transitions. This does not, however, guarantee that the optimal solution is found, as that would require testing all permutations.

In order to further reduce the number of iterations required to optimize the merging result, we have leveraged an additional heuristic that allows us to get very good results by only testing a smaller computationally tractable number of the possible permutations. This "random" heuristic consists in picking a fixed number of random permutations and merging them, returning the sequence from the set that contains the lowest number of transitions in the merged graph. For our

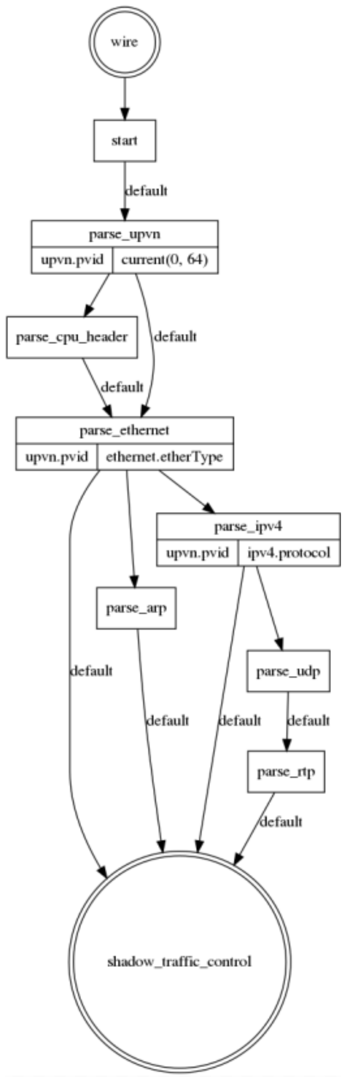


Figure 5.9: Parse graph resulting from the merge of $P10$ followed by $P6$.

testing set, we merged 200 random sequences (20 tests with ten sequences each), and arranged the results into a plot, shown in Figure 5.11. This test produced an output in less than ten minutes, halving the time required by only using the other heuristic. As it can be seen in the plot, merged sequences that produce a number of transitions equal to 40 are common, and thus this heuristic can easily find the best sequence produced by our aggregation heuristic. This heuristic does not, however, guarantee that the optimal sequence is found, and becomes less effective as the number of input programs increases, but it allows us to quickly get interesting improvements.

5.4 Summary

In this chapter, we first described the testing environment for our solution, in Section 5.1. Then, we illustrated the mechanism used to verify P4Visor++'s ability to guarantee *correctness* and *isolation*, in Section 5.2. To conclude, in Section 5.3, we presented the results achieved by our

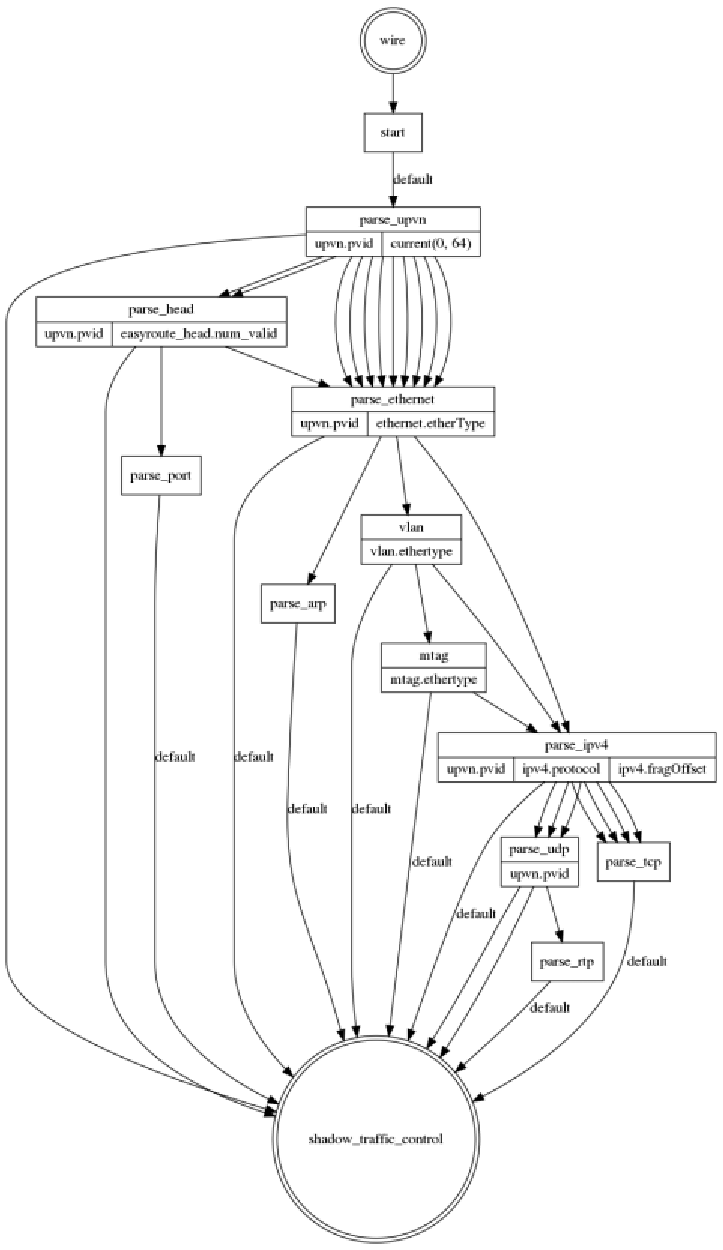


Figure 5.10: Optimal merge result achieved by merging our entire program set.

system, and compared them with the state-of-the-art in PDP virtualization.

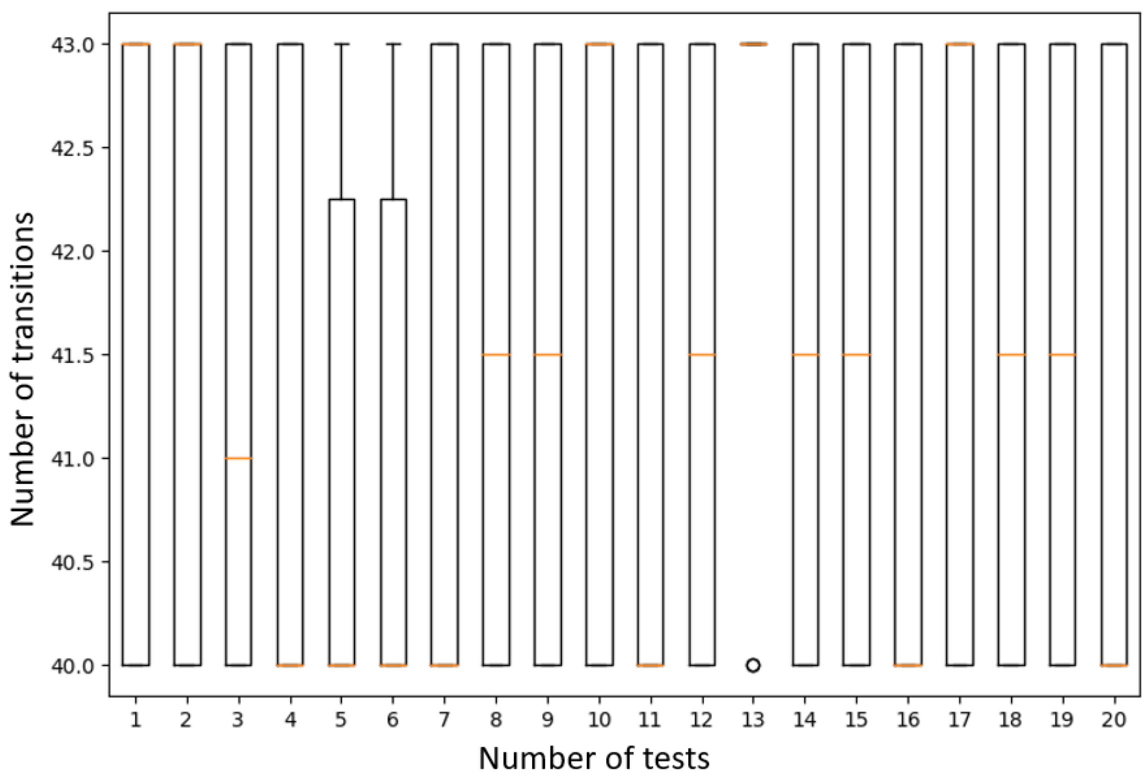


Figure 5.11: Boxplot showcasing the result of merging a fixed number of random, different sequences from our testing set. This plot shows that, for our set, the optimal merging result (i.e., a merged graph containing 40 transitions) can be achieved through multiple different sequences, and can therefore be found by testing only a small subset of the permutations.

Chapter 6

Conclusion & Future Work

As PDPs become increasingly popular, the development of virtualization techniques that enable flexibility in the deployment of custom network functionalities onto these targets and the possibility to share these hardware resources to increase their utilization, for instance in cloud environments, becomes a necessity. We have analyzed the two mainstream approaches for the virtualization of PDPs, namely Emulation-Based and Code Merging. We concluded that the former approach requires large resource overheads, making it impractical to deploy, as such resources are usually scarce in the current state-of-the-art PDP switch targets. The state-of-the-art code merging approach, P4Visor, significantly improves the efficiency of PDP virtualization over emulation-based systems. However, it has several limitations that may also preclude its deployment in practice, namely, its inability to merge more than two programs, the requirement of a high degree of similarity between the merged programs and the inefficiency in total amount of resources used.

In this thesis, we have improved over the state-of-the-art with P4Visor++: a system that enables a more flexible and efficient merging of P4 programs, while guaranteeing *isolation* and *correctness*. As the main contributions we introduced the ability to merge a potentially unrestricted number of input P4 programs, and an innovative mechanism to reduce resource usage in the merged program, by efficiently and correctly sharing equivalent code portions across multiple programs. Our solution focuses on the parse graph stage of the P4 programs, establishing equivalences between headers and parser states from different input programs and combining them into a larger merged graph, where some nodes are shared by multiple programs. This task is challenging due to the restrictions imposed by the P4 language specification and by physical constraints of some target architectures. Our solution shows significant efficiency improvements in terms of resources used for the parser stage in the merged program.

In future work, we plan to integrate code merging techniques to achieve better resource sharing across the other blocks of a PDP target, namely the MAT stages of PISA switches, and to port this solution to the $P4_{16}$ version of the language. We also intend to investigate the possibility to share stateful memory in PDP targets across the different merged programs. Finally, we plan to investigate the possibility of performing *seamless reconfiguration*, that is, removing and adding programs to the PDP target without the need to restart it. This may be achieved by exploring the way resources are represented in the switch's hardware, allowing for specific segments to be "turned on and off", and consequently removing and adding programs without disrupting the other programs' ability to process the incoming traffic.

Bibliography

[1] Diego Kreutz, Fernando MV Ramos, Paulo Esteves Verissimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103(1):14–76, 2014.

[2] Product brief tofino page — barefoot. `https://barefootnetworks.com/products/brief-tofino/`, Last accessed on 2020-08-29.

[3] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. *ACM SIGCOMM Computer Communication Review*, 43(4):99–110, 2013.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[5] Naga Katta, Mukesh Hira, Changhoon Kim, Anirudh Sivaraman, and Jennifer Rexford. Hula: Scalable load balancing using programmable data planes. In *Proceedings of the Symposium on SDN Research*, pages 1–12, 2016.

[6] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, pages 35–49. ACM, 2016.

[7] Cheng Zhang, Jun Bi, Yu Zhou, and Jianping Wu. Hypervdp: High-performance virtualization of the programmable data plane. *IEEE Journal on Selected Areas in Communications*, 37(3):556–569, 2019.

[8] Peng Zheng, Theophilus Benson, and Chengchen Hu. P4visor: Lightweight virtualization and composition primitives for building and testing modular programs. In *Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies*, pages 98–111. ACM, 2018.

[9] Hardik Soni, Thierry Turletti, and Walid Dabbous. P4bricks: Enabling multiprocessing using linker-based network data plane architecture. 2018.

[10] Changhoon Kim. Programming the network data plane: What, how, and why? https://conferences.sigcomm.org/events/apnet2017/slides/chang.pdf, Last accessed on 2020-09-08.

[11] p4runtime. https://github.com/p4lang/p4runtime. Accessed: 2020-08-29.

[12] P4visor's github repository. https://github.com/Brown-NSG/P4Visor. Accessed: 2020-08-29.

[13] Xiangyu Gao, Taegyun Kim, Michael D Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. Switch code generation using program synthesis. In *Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication*, pages 44–61, 2020.

[14] p4c. https://github.com/p4lang/p4c. Accessed: 2020-08-29.

[15] bmv2. https://github.com/p4lang/behavioral-model. Accessed: 2020-08-29.

[16] Hlir github repository. https://github.com/p4lang/p4-hlir. Accessed: 2020-08-29.

[17] Simple router. https://github.com/P4-vSwitch/vagrant. Accessed: 2020-08-29.

[18] Flowlet. https://github.com/CS344-Stanford-18/p4-mininet-tutorials/tree/master/SIGCOMM_2015/flowlet_switching/p4src. Accessed: 2020-08-29.

[19] Heavy hitter. https://github.com/CS344-Stanford-18/p4-mininet-tutorials/tree/master/SIGCOMM_2016/heavy_hitter. Accessed: 2020-08-29.

[20] Portknock firewall. https://github.com/signorello/myP4repo/tree/master/portKnockFirewall. Accessed: 2020-08-29.

[21] Mc nat. https://github.com/NEOAdvancedTechnology/mc_nat_P4. Accessed: 2020-08-29.

[22] Timestamp. https://github.com/NEOAdvancedTechnology/ts_switching_P4. Accessed: 2020-08-29.

[23] Ecmp. https://github.com/ccascone/onos-p4-dev. Accessed: 2020-08-29.

[24] mtag edge. https://github.com/p4lang/p4-spec/tree/master/p4-14/v1.0.2/mtag-example. Accessed: 2020-08-29.

[25] Source routing. https://github.com/p4lang/p4c/blob/master/testdata/p4_14_samples. Accessed: 2020-08-29.

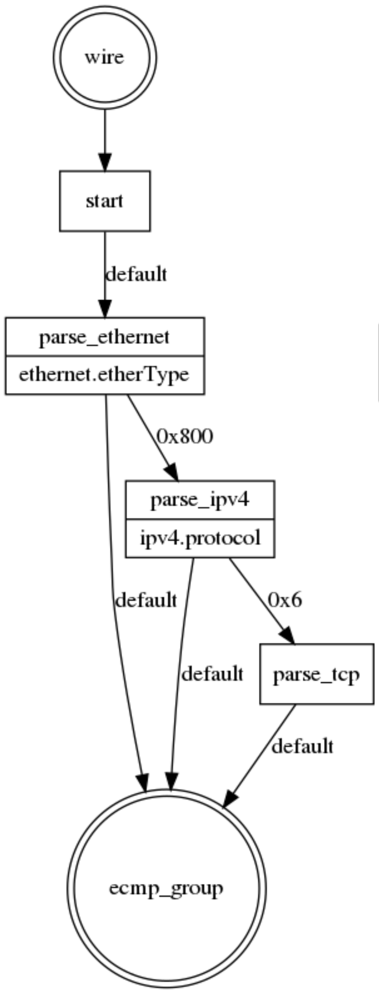
[26] Simple router w/ arp. https://github.com/p4lang/PI/tree/master/proto/demo_grpc. Accessed: 2020-08-29.

[27] Scapy. https://scapy.net/. Accessed: 2020-08-29.

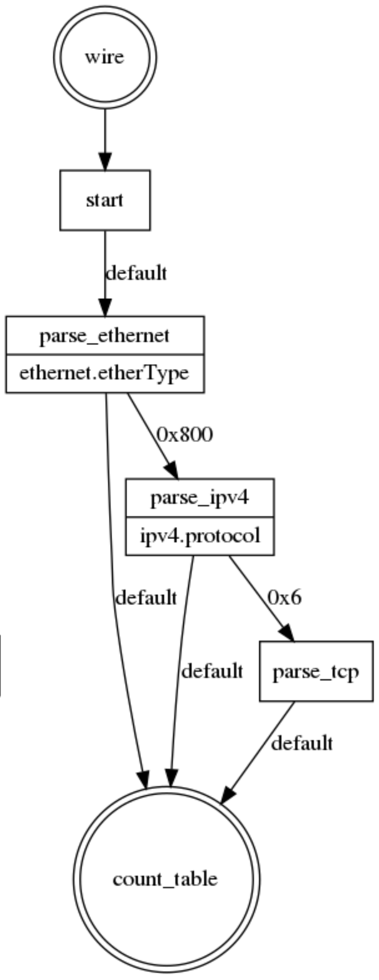
[28] Shan Sinha, Srikanth Kandula, and Dina Katabi. Harnessing tcp's burstiness with flowlet switching. In *Proc. 3rd ACM Workshop on Hot Topics in Networks (Hotnets-III)*. Citeseer, 2004.

Appendix A

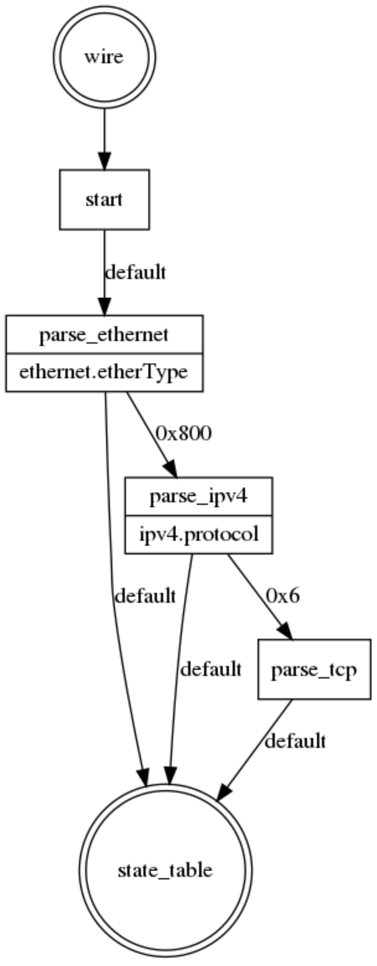
Parse graphs of all the programs



(a)



(b)



(c)

Figure A.1: Programs merged in Test A - (a) P2; (b) P3; (c) P4

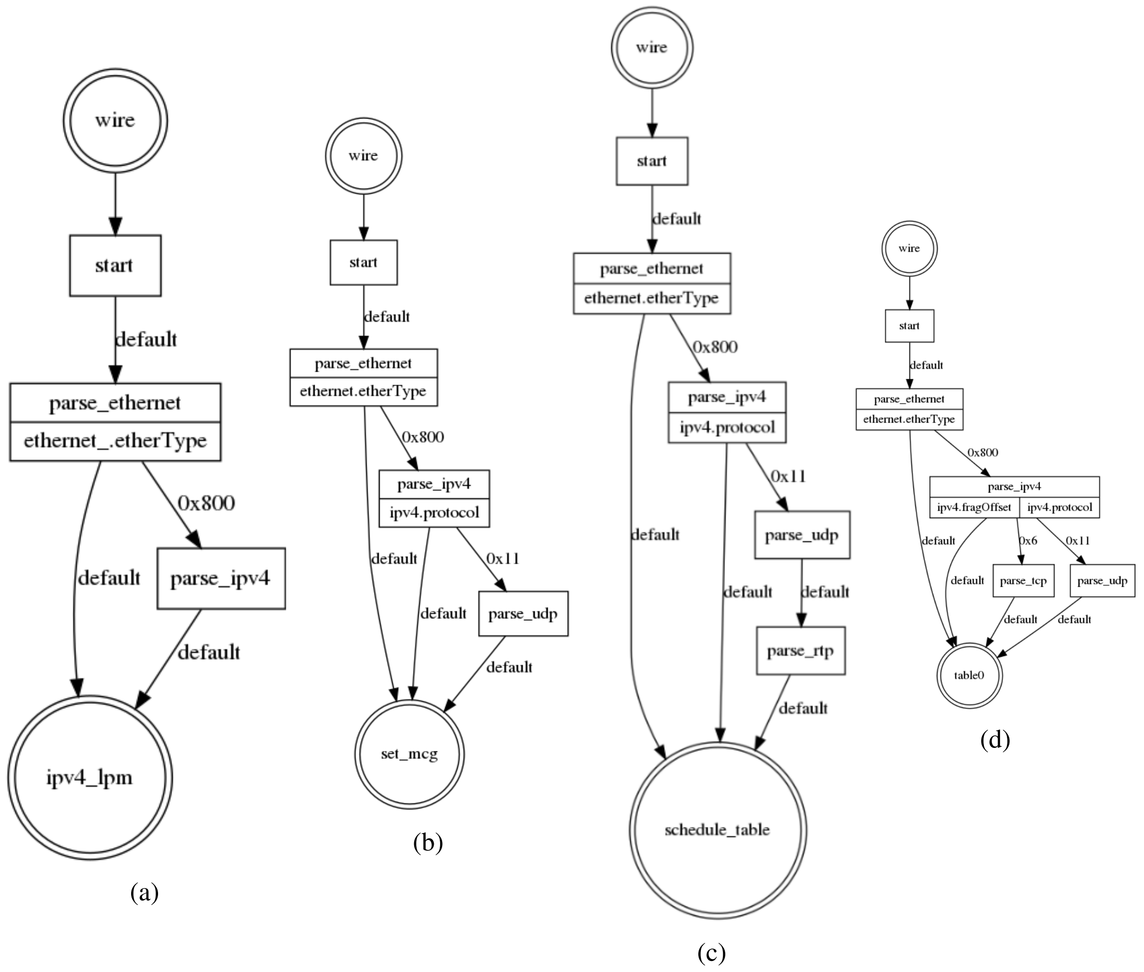


Figure A.2: Programs merged in Test B - (a) P1; (b) P5; (c) P6; (d) P7

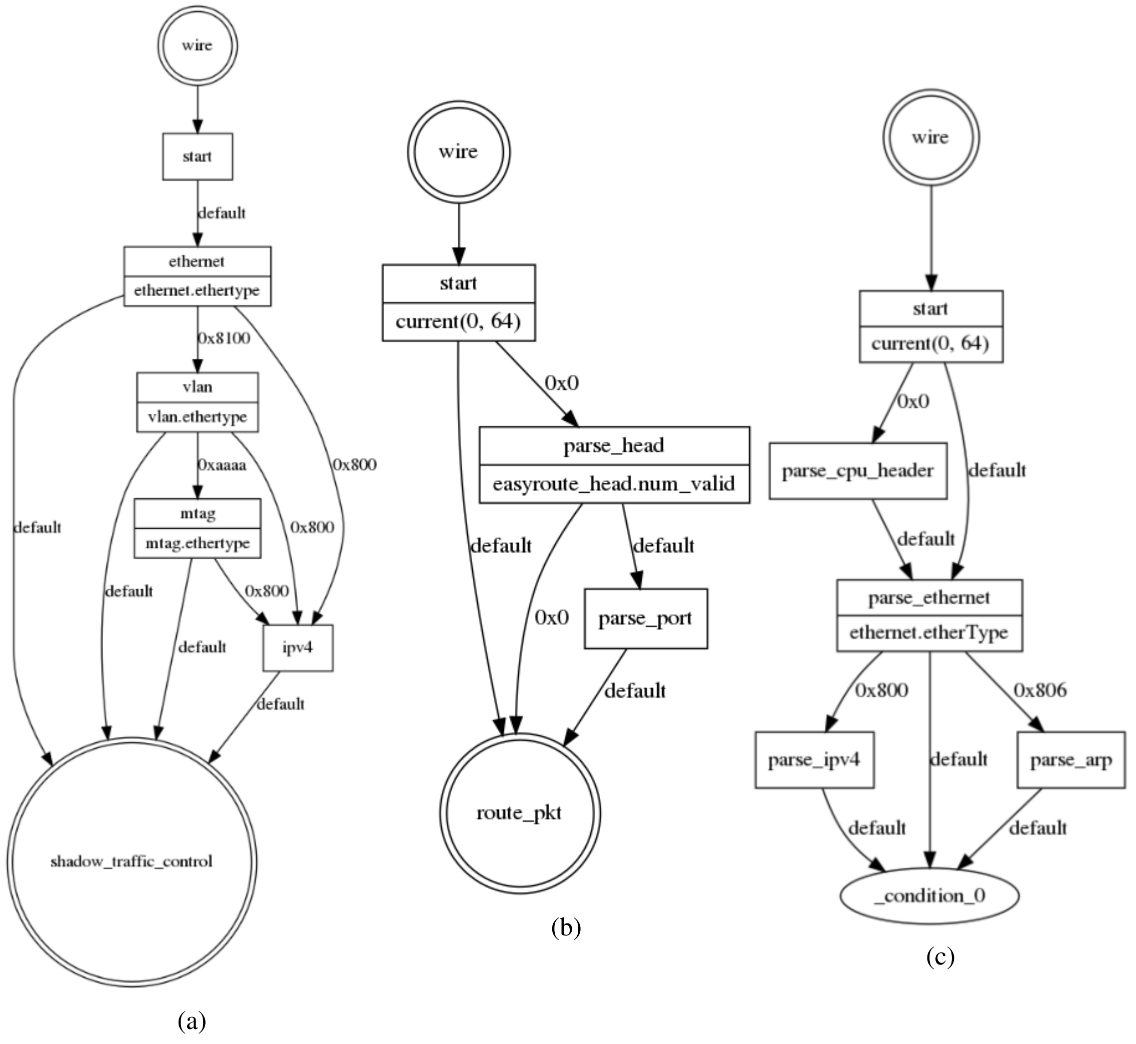


Figure A.3: Programs merged in Test C - (a) P8; (b) P9; (c) P10

Appendix B

P4visor++ Documentation

P4Visor++ is a tool used to efficiently merge the parsing stage of multiple P4 programs. The merging process results in the generation of a compiled JSON file, which contains the merged parse graph, where the graphs from each input program are correctly represented and isolated. The compiled JSON file can then be installed on the Bmv2 P4-capable software switch.

All of the software developed for this work is available at:
https://github.com/netx-ulx/P4Visor/tree/dsequeira.

B.1 Requirements

In order to merge P4 programs and to test the result with the following instructions, the user must first install the following dependencies:

- - *[p4c-bm]* (https://github.com/p4lang/p4c-bm)
- - *[bmv2]* (https://github.com/p4lang/behavioral-model)

B.2 Merging P4 Programs

B.2.1 Interface

To merge multiple programs, we leverage the Python script created by P4Visor (https://github.com /Brown-NSG/P4Visor), `ShadowP4c-bmv2.py`, with some modifications.

The script must be used with the following input arguments:

- The first program to be added to the merged program:
  - `--shadow_source *path_to_p4_program*`
- The second program to be added to the merged program:
  - `--real_source *path_to_p4_program*`
- The path of any additional program to be merged (separated by spaces, if more than one):

  - `--l *path_to_p4_program* ... *path_to_p4_program*`

- The name of the output JSON file:
  - `--json_mg *path_to_dir_with_name.json*`
- The option to generate a visual representation of the graphs:
  - `--gen-fig`
- The directory to where the output files will be stored:
  - `--gen_dir *path_to_dir*`
- The mode of operation (must always be Diff-Testing):
  - `--Diff-testing`

The execution of this script will generate the merged JSON file and the visual representation of the graphs and store them into the directory specified with `--gen_dir`. Additionally, a file named `evalFinal.txt` will be created and stored at the project's root directory, containing useful information regarding the amount of resources used by the parser graph in the merged program.

B.2.2 Merge Example

To illustrate how the merging of multiple programs is achieved, we provide the following example which merges three P4 programs (flowlet.p4, portKnockFirewall.p4, heavy_hitter.p4, which are available under the folder 'tests/testAll/'). We first create a directory on the project's root, called `example`, wherein we copy our three programs. Afterwards, to merge the programs, we use the following command in a terminal opened at the level of the project's root directory:

-
  ```
  python ShadowP4c-bmv2.py --real_source example/portKnock-
  Firewall.p4 --shadow_source example/flowlet.p4 --json_mg
  example/merged.json --l example/heavy_hitter.p4 --gen-fig
  --gen_dir example --Diff-testing
  ```

The merged JSON file will be placed in the `example` folder, under the name `merged.json`.

B.3 Reproducing the results of 'Code Merging for Programmable Data Plane Virtualization'.

P4Visor++ has been developed within the framework of an MSc. thesis carried out by Duarte Sequeira at the Faculty of Sciences of the University of Lisbon in 2020. In order to evaluate that work, three different sets of P4 programs, showing different degrees of similarity, were merged. The programs in those sets are all available under the folder 'tests/testAll/'.

To recreate those tests, the following commands must be executed:

- Test A:

  -
    ```
    python ShadowP4c-bmv2.py --real_source tests/testAll/portKnockFirewall.p4 --shadow_source tests/testAll/flowlet.p4 --json_mg tests/testAll/merged.json --l tests/testAll/heavy_hitter.p4 --gen-fig --gen_dir tests/testAll --Diff-testing
    ```
- Test B:

  -
    ```
    python ShadowP4c-bmv2.py --real_source tests/testAll/mc_nat.p4 --shadow_source tests/testAll/ecmp.p4 --json_mg tests/testAll/merged.json --l tests/testAll/simple_router.p4 tests/testAll/timestamp.p4 --gen-fig --gen_dir tests/testAll --Diff-testing
    ```
- Test C:

  -
    ```
    python ShadowP4c-bmv2.py --real_source tests/testAll/mtag-edge.p4 --shadow_source tests/testAll/source_routing.p4 --json_mg tests/testAll/merged.json --l tests/testAll/simple_router_with_arp.p4 --gen-fig --gen_dir tests/testAll --Diff-testing
    ```
- Additionally, the merge sequence for all the programs in our test-set is the following:

  -
    ```
    python ShadowP4c-bmv2.py --real_source tests/testAll/simple_router_with_arp.p4 --shadow_source tests/testAll/source_routing.p4 --json_mg tests/testAll/merged.json --l tests/testAll/timestamp.p4 tests/testAll/mtag-edge.p4 tests/testAll/portKnockFirewall.p4 tests/testAll/heavy_hitter.p4 tests/testAll/simple_router.p4 tests/testAll/ecmp.p4 tests/testAll/mc_nat.p4 tests/testAll/flowlet.p4 --gen-fig --gen_dir tests/testAll --Diff-testing
    ```

B.4 Contacts

If you have any questions, you can reach me (Duarte Sequeira) at dudaxsek97@gmail.com