UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Juán Sebastián Mejía Vallejo

Design, Implementation and Evaluation of a Broadband Network Gateway using a Programmable Packet Processor

Projeto, implementação e avaliação de um gateway de rede de banda larga usando um processador de pacote programável

CAMPINAS
2018

Juán Sebastián Mejía Vallejo

Design, Implementation and Evaluation of a Broadband Network Gateway using a Programmable Packet Processor

Projeto, implementação e avaliação de um gateway de rede de banda larga usando um processador de pacote programável

Dissertation presented to the Faculty of Electric and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Electric Engineering in the area of Computer Engineering..

Dissertação apresentada à Faculdade de Engenharia Elétrica e de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação.

Supervisor/Orientador: Prof. Dr. Christian Rodolfo Esteve Rothenberg

Este exemplar corresponde à versão final da Dissertação defendida por Juán Sebastián Mejía Vallejo e orientada pelo Prof. Dr. Christian Rodolfo Esteve Rothenberg.

CAMPINAS
2018

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Rose Meire da Silva - CRB 8/5974

M479d

Mejía Vallejo, Juán Sebastián, 1985-
Design, implementation and evaluation of a broadband network gateway using a programmable packet processor / Juán Sebastián Mejía Vallejo. – Campinas, SP : [s.n.], 2018.

Orientador: Christian Rodolfo Esteve Rothenberg.
Dissertação (mestrado) – Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica e de Computação.

1. Virtualização de redes. 2. Redes de computadores. 3. Redes definidas por software (Tecnologia de rede de computador). I. Esteve Rothenberg, Christian Rodolfo, 1982-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: Projeto, implementaçâo e avaliação de um gateway de rede de banda larga usando um processador de pacote programável
Palavras-chave em inglês:
Network function virtualization
Computer networks
Software defined networking (Computer network technology)
Área de concentração: Automação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Christian Rodolfo Esteve Rothenberg [Orientador]
Fabio Luciano Verdi
Leonardo de Souza Mendes
Data de defesa: 18-12-2018
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-8006-8296
- Currículo Lattes do autor: http://lattes.cnpq.br/3382339693358283

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Juán Sebastián Mejía Vallejo RA: 152443
Data da Defesa: 18 de dezembro de 2018
Título da Tese: "Projeto, Implementação e Avaliação de um Gateway de Rede de Banda Larga Usando um Processador de Pacote Programável".

Prof. Dr. Christian Rodolfo Esteve Rothenberg (Presidente)
Prof. Dr. Fabio Luciano Verdi
Prof. Dr. Leonardo de Souza Mendes

A ata de defesa, com as respectivas assinaturas dos membros da Comissão Julgadora, encontra-se no SIGA (Sistema de Fluxo de Dissertação/Tese) e na Secretaria de Pós-Graduação da Faculdade de Engenharia Elétrica e de Computação.

When a goal is reached, it is necessary to think and value those things that led us to achieve it; the path walked, those people who supported us, difficulties, joys, and the lessons learned. I want to dedicate this work to all the people who gave me their prayers, words of encouragement, but also there wisdom, love, and support; this goal has been possible thanks to all of you. I am sure that in any place God will allow us to meet people like you and to keep growing both personal and professionally.

Acknowledgements

My thanks to my parents Jaime and Maria Teresa, my sisters, family, and whoever has motivated me to go further by giving the example of dedication and best effort.
I am grateful for my friends in Campinas, Rayanna Amorim for taking care of me in awkward moments, Jorge Fernandez for his friendship, Amparo Diaz for her motivation to go forward, William Diaz, Gustavo Perdomo, and his family, Pablo Florez and all the people that made life lighter.
Christian Esteve Rothenberg, my advisor, and friend who motivated and taught me the SDN concepts.
Many thanks to the INTRIG group, Nathan Saraiva, Suneet Singh, Gyanesh Patra, Fabricio Rodriguez, Alex Lachos and all the group for his friendship and support many times outside the university.
Thanks to the people of the DCA laboratory for inviting me to lunch everyday.
Thanks to the Ericsson Innovation Center Brazil and Funcamp under grant agreement 4881.3, for sponsoring this work, and the Ericsson Traffic Lab in Hungary, for technical support.
Finally, my sincere acknowledgments to those people who have collaborated with code, bug reports or suggestions.

"A ship in port is safe. But that's not what ships were built for"
John Augustus Shedd

Abstract

Broadband Network Gateway (BNG), also known as Broadband Remote Access Server (BRAS) plays a crucial role in today's internet, as it handles the majority of access network traffic, implementing network policies and services that a Internet Service Provider (ISP) defines per subscriber. However, this network device is normally expensive, proprietary, limited and slow to upgrade, and a point of failure when deploying new functionalities and correcing issues on the network without disrupting normal service operations.

This work intends to design, implement and evaluate a flexible and optimized BNG Software (S/W) switch by using programming protocol-independent packet processors (P4) language, taking advantage of its features to describe the packet processing with target-agnostic data-plane programmability which is advantageous by providing to developers (i.e., network operators) an alternative to packet handling schemes.
We propose to use the Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD) compiler which merges the P4 abstraction and OpenDataPlane (ODP) APIs to support Linux-based network applications with high performance across some architectures (ARM, Intel, MIPS, and PowerPC). The actual version of MACSAD brings support for the previous release of the P4 language (P4_14). Therefore part of this work is to bring support to MACSAD for the current release (P4_16) on which the proposed BNG dataplane is built. We use functional and performance evaluation of the BNG S/W switch in a realistic scenario by using both hardware (H/W) and S/W open source traffic generators. The results show the impact of both target parameters such as the number of cores, burst sizes, packet IO, and workloads and how it affects the performance regarding throughput and latency bringing the best parameter configuration for our implementation.

Keywords: Computer Networks; Software-Defined Networking; OpenFlow; Programmable Dataplane; Performance Evaluation; Traffic Generators; Packet IO.

Resumo

BNG, também conhecido como BRAS desempenha um papel crucial na Internet atual, ele maneja a maioria do tráfego da rede de acesso, implementando políticas e serviços que um Internet Service Provider (ISP) define por assinante. Porém, este dispositivo de rede é caro, proprietário, limitado e de lenta atualização, virando um ponto de falha para implantar novas funcionalidades e corrigir problemas na rede sem interromper a operação normal do serviço.

Este trabalho pretende projetar, implementar e evaluar um BNG S/W switch flexível e otimizado usando a linguagem P4, aproveitando seus recursos para descrever o processamento de pacotes com programação de plano de dados agnóstica ao alvo que é vantajoso ao fornecer aos desenvolvedores (por exemplo, operadores de rede) uma alternativa aos esquemas de manipulação de pacotes.
Nos propomos usar o compilador MACSAD que junta a abstração de P4 e APIs de OpenDataPlane (ODP) para suportar aplicativos de rede baseados em Linux com alto desempenho em varias arquiteturas (ARM, Intel, MIPS e PowerPC). A versão atual do MACSAD dá suporte para a versão anterior da linguagem P4 (P4 _14). Portanto, parte deste trabalho é dar suporte ao compilador MACSAD para o versão atual (P4 _16) no qual é construído o plano de dados de BNG. Além disso, foi feita uma avaliação funcional e de desempenho do BNG S/W switch em um cenário realista e usando dois geradores de tráfego Open Source em H/W e S/W. Os resultados mostram o impacto dos parâmetros do dispositivo alvo, tais como o número de núcleos, tamanhos de burst, pacotes de E/S e cargas de trabalho que afetam o desempenho em termos de taxa de transferência e latência fornecendo os melhores parametros de configuração para nossa implementação.

Palavras-chaves: Redes de Computadores; Software Defined Networking; OpenFlow; Plano de dados programável; Avaliação de desempenho; Geradores de Tráfego; Pacotes de E/S.

List of Figures

List of Tables

Acronyms and Abbreviations List

API Application Programming Interface

ARP Address Resolution Protocol

ASIC Application Specific Integrated Circuit

BNG Broadband Network Gateway

BRAS Broadband Remote Access Server

CPE Customer Premise Equipment

DL Download Link

DMA Direct Memory Access

DPDK Data Plane Development Kit

DSL Digital Subscriber Line

DUT Device Under Test

FIB Forwarding Information Base

GRE Generic Routing Encapsulation

H/W Hardware

HLIR High-Level Intermediate Representation

ICMP Internet Control Message Protocol

IP Internet Protocol

IPG Inter-Packet Gap

IPv6 Internet Protocol Version 6

ISP Internet Service Provider

JSON JavaScript Object Notation

LPM Longest Prefix Match

MAC Media Access Control

MACSAD Multi-Architecture Compiler System for Abstract Dataplanes

Mpps Mega packets per second

NAT Network Address Translation

NFPA Network Function Performance Analyzer

NFV Network functions virtualization

ODP OpenDataPlane

OF OpenFlow

ONF Open Network Foundation

OS Operating System

OSNT Open Source Network Tester

OVS Open vSwitch

P4 programming protocol-independent packet processors

QoS Quality of Service

RAM Random Access Memory

S/W Software

SDK Software Development Kit

SDN Software Defined Networking

TCP Transmission Control Protocol

TTL Time to live

UL Upload Link

VM Virtual Machine

VoD Video-on-Demand

VoIP Voice Over Internet Protocol

Contents

Chapter 1

Introduction

Internet services such as Video-on-Demand (VoD), IP telephony (VoIP), Online games and cloud-based services are increasingly offered over broadband networks (OECD, 2016). These internet services carry new challenges for the Internet Service Providers (ISPs) to deploy novel infrastructure technologies on the core and access network, enhancing the network performance while also modifying certain aspects of traditional systems.
The Broadband Network Gateway (BNG) plays a key role on an ISP network. It has the functions of managing all the traffic from access networks (e.g., Digital Subscriber Line (DSL) traffic), allowing authentication for thousands of subscribers, monitoring, establishing sessions, tunnels (e.g., Generic Routing Encapsulation (GRE)), packet forwarding and user rate policing. It means that it is a single point on which is added all the network traffic which brings the performance and Quality of service (QoS) for all the network users.
Therefore, it is no surprise that the BNG has become an expensive hardware proprietary box on which the operator has to pay for some functionalities that won't be used and wait a long time until the next version is available to upgrade its hardware and software.
One strategy to address the issues above is to introduce intelligence in the infrastructure by designing a flexible and programmable network. The recent Software-Defined Networking (SDN) technology enables an operator to implement more flexible networks allowing direct programming of network control functions. This technology breaks the vertical integration of networks (KREUTZ *et al.*, 2015) by splitting the forwarding plane of switches between control (network's control logic) and data plane (forwarding devices) as shown in Figure 1.1.

In the last years, programmable switching chips have become more used, along with new packet processing languages such as Programming Protocol-Independent Packet Processors (P4) and Huawei's Protocol Oblivious Forwarding (POF) (JOSE *et al.*, 2015). P4 has been proposed to overcome OpenFlow (OF) programming limitations by enabling the description of packet forwarding and configuring any network protocol in any target (P *et al.*, 2013). Compared to OF, it provides a higher level of abstraction for programming the network and offers wide community support.

In addition, there are different P4 compilers for S/W switch programming such as P4C (P4C, 2018), Multi-Architecture Compiler System MACSAD (PATRA *et al.*, 2017), T4P4S (VOROS *et al.*, 2018) and H/W switch programming such as Xilinx SDNet (XILINX,)

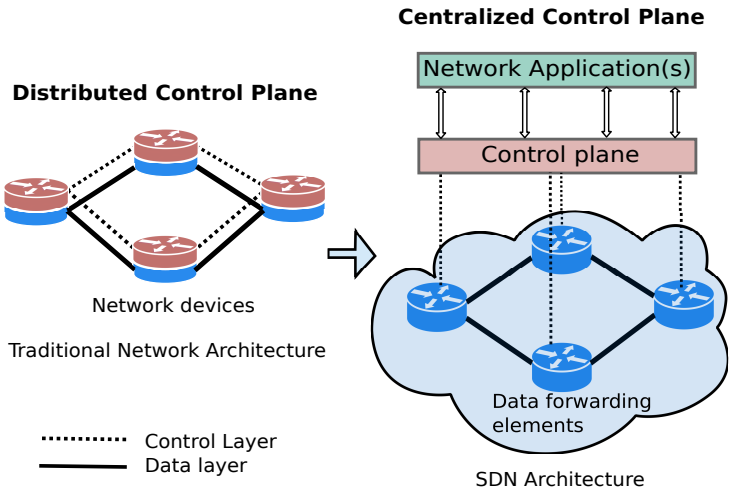


Figure 1.1: Traditional networking vs SDN model.

for FPGA and Capilano Software Development Environment (SDE) (BAREFOOT, 2018) for ASICs.

In this work, we present an approach to implement a P4-based BNG S/W switch to describe the dataplane built on top of the MACSAD compiler which eases its deployment by bringing three main capabilities: programmability, portability and high performance in the data plane and control plane.

On the other hand, the novel technologies require network researchers to asses these new implementations and investigate their performance in terms of network metrics. We present two traffic generator tools which can enable this process. The first tool, Network Function Performance Analyzer (NFPA), is an open source traffic generator which supports a fully automated workflow for performing and evaluating network experiments. One of its key features is that it can be run in any server generating customized network traffic, foster the test network environment and gain reproducible experiment results. The second one is Open Source Network Tester (OSNT) an open source hardware-based traffic generator, which delivers higher accuracy metrics and automated workflow to perform network experiments.

1.1 Problem Statement

The broadband networks are requiring novel solutions in their architecture and infrastructure to increase the data rate, reduce latency and improve reliability while optimizing power consumption. Therefore, the design and operation of the transport network must consider the QoS levels requirements for different services. However, the legacy network devices don't bring enough flexibility to face the needs of the next generation of networks.

Moreover, as mentioned above the BNG network device is hardware-based, therefore to turn this rigid hardware device into a software-based network and reduce time to market new services these functionalities must be decomposed and dynamically instantiated at specific points of the network according to the network architecture. This concept is following the trend of SDN and NFV that turn some network functionalities into virtualized

software processing running on a server (e.g., off-the-rack x86 servers), switches or even cloud computing infrastructure (HAN *et al.*, 2015).
Nowadays, we find some programmable data plane devices enabled to define some aspects including packet parsing, actions, and the layout of packet processing on the hardware device itself. Despite the increased programmability that these devices offer, the target architecture that defines some resources usage such as many processor cores, memory and Network interface cards, restricts the BNG device features to be deployed on a commodity server.
Our approach is to create a fully open software BNG forwarding device using the lastest version (16 versions released since May of 2017) of programming protocol-independent packet processors (P4) a high-level description language which offers a more stable (future-proof) and more straightforward data plane definition compared to the previous one P4_14. Such P4 language can customize the forwarding behavior of the switch and may also bring portability to other hardware or software switches that support the same language.
This use case is intended to be implemented with MACSAD which eases the implementation over many S/W and H/W targets. To implement a BNG S/W switch some issues arise:

1. How to design a functional implementation of a BNG S/W switch in P4_16 to describe the data plane functionalities?

2. How to bring the support of P4_16 in MACSAD? The current version of MACSAD is based on the P4_14 version, hence, to compile and implement the virtualized BNG S/W switch it is mandatory to update it to the P4_16 version.

3. How to map the P4 functions with ODP APIs on MACSAD? The ODP helper library provides the necessary support for the forwarding dataplane. However, there is not an actual implementation of MACSAD to enable Encap/Decapsulation and Drop functions. Hence, it is necessary to add these modules to support our implementation.

4. How to evaluate the performance of the BNG S/W switch? Assessment of the BNG packet processing is indispensable. We propose a performance evaluation regarding network metrics such as throughput (packets per second) and packet delay (milliseconds), to check the BNG performance and how affected the reliability is at high rates.

1.2 Objectives

The main objective of this work is to design, implement and evaluate the BNG S/W switch using a Programmable Packet Processor. To address the issues described in the problem statement, we propose the following specific objectives:

- Design the architecture and functional implementation of our BNG S/W switch. Define the P4 workflow and develop the P4 program to create the dataplane relying on the latest version of P4 passing the compilation procedures.

- Bring P4_16 support for MACSAD. The MACSAD compiler generates a C-based dataplane code which is compiled and executed in different targets. According to this, we need to use the HLIR16 project and its dependencies such as P4C compiler to create an intermediate representation (IR) that brings the data structure used as the code base in the MACSAD core to translate the P4 code into C code.

- Map the P4 functions used in the implementation of BNG with ODP APIs on the MACSAD. This procedure adds support to the BNG functionalities such as Encap/Decap and Drop functions. On the other hand, the controller has to interact with MACSAD to manage table action management; hence this interface should be added as well.

- Evaluate the BNG performance with S/W and H/W traffic generator tools configuring a realistic network scenario with commodity servers on which the BNG S/W switch will run. Thus, in order to asses the BNG packet processing capabilities, we need to define the packet traces, workloads and adequate use case scenarios.

1.3 Contributions

The main contributions of this thesis unfold from the achievements of the above-mentioned objectives. More specifically, the most notable contributions are, to the best of our knowledge, (1) the first public open source design and implementation of a BNG-like pipeline using the P4 language, and (2) a method to measure the latency of network devices through Encapsulation/De-capsulation by adding a new time-stamp packet header, which is parsed along with the other headers in the programmable packet processor.

As the main tangible result, our BNG prototype implementation in P4/ODP over an x86 platform implements a set of data plane and control plane functionalities in a cost-effective and flexible way. Table 1.1 summarizes the functionalities of our BNG implementation and compares them with a commercial product (NETWORKS, 2018). This comparison is not meant to be complete or exhaustive but mainly serves as a proof point of our feature coverage and, more generally, of the potential cost-performance benefits of open source dataplane designs and commercial off-the-shelf (COTS) server technologies.

Function		Our P4 BNG	Juniper BNG (MX-150)
Routing protocols	L2	1Million.	x
	L3 (IPv4)	1Million.	37 Millions.
	L3 (IPv6)		x
	ARP	x	x
	IGMP		x
	TCP	x	x
	MSDP		x
	MPLS		x
Tunneling	GRE	x	x
	PPoE		x
	L2TP		x
Traffic management policies	Rate limiter	x	x
	Scheduler		x
Configuration interfaces	Radius (Autentication)		x
	DHCP		x
	SDN control plane	x	
	VLAN		x
Security	Firewall	x	x
Network interfaces		4x SFP 1/10 GbE port	8x 1GbE, 4x SFP 1/10GbE, 1x RJ45 ports
Number of user sessions		0.1Million	1 Million
Cost estimate		∼ US$ 1000	∼US$ 15000

Table 1.1: Feature comparison of our P4 BNG implementation with a commercial H/W BNG.

1.4 Outline

In the introduction, we described the motivation and challenges that justify this work. Hereby, we present the rest of this work which has been structured in five main chapters as follows. In Chapter 2, we present the Background in the context of BNG S/W switch implementation, describing the involved technologies, tools, compilers, and languages on which this work relies. Furthermore, we present a literature review which analyzes the related works and compares them with our work. In Chapter 3, we describe the design methodology and implementation aspects of the BNG S/W switch on which we explain all the modules and its relationship between them. Thus, we describe relevant aspects across the dataplane and the BNG controller. In Chapter 4, we present the functional and performance evaluation as well as the results. Finally, Chapter 5 highlights the results and issues found in the implementation process and discusses future works.

Chapter 2

Background and Related Work

In this section, we present the main concepts of BNG, P4 language, and tools used in the workflow such as the MACSAD compiler, OpenDataPlane (ODP), a review of two traffic generators used in our project and finally we compare our implementation with related works.

2.1 Broadband Network Gateway (BNG)

The first generation networks relied on centralized BRAS routers and were driven by the customer demand for High-speed Internet (HSI). The second generation of Ethernet-based BNG routers was driven by subscriber demand for a linear TV service (content broadcast at specific times, e.g., Netflix) delivered along with voice and HSI services (ALCATEL-LUCENT, 2010).
The Customer Premise Equipment (CPE) represents the triple play communications devices: Telephone (Voice), PC (Internet) and the Set-top box (TV). However all these devices are connected to a Home Gateway (HG) which brings the interface with the network through an access technology like DSL.
One or more HG's can be connected to a single DSLAM that sends the traffic towards the Internet via BNG and edge routers and provides connectivity to the ISP subscriber (See Figure 2.1).

The broadband network provides connectivity, authentication, applications, services,

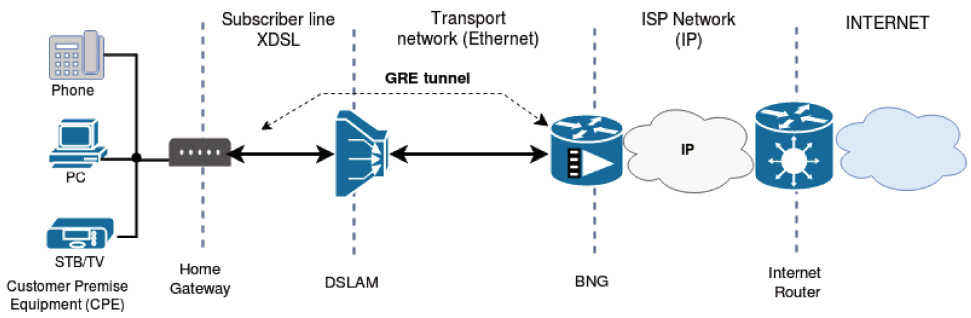


Figure 2.1: Broadband access network architecture.

and network policies to its subscribers; therefore, these procedures involve the premise of

session establishment using access communication protocols which are managed in the BNG. The most common protocols to establish sessions are:

- The PPP over Ethernet (PPPoE): Uses the point-to-point (PPP) protocol.
- The IP over Ethernet (IPoE): Uses the IP protocol that runs between CPE and BNG.
- Generic Routing Encapsulation (GRE): encapsulation protocol brings virtual Point-to-Point connections through the IP network.

An BNG centralizes functions such as:

- Session management and header en/de-capsulation.
- Interface to Authorization, Authentication and Accounting services.
- Address Resolution Protocol (ARP) proxy to manage the requests from the network interface on the BNG side.
- Network Address Translation (NAT) to route the packets towards the operator's core Network.
- QoS enforcement.

In the next section, we will describe the main advantages, architecture and compilers of the P4 programming language to implement the functionalities of our packet processor as well as MACSAD; a P4 compiler which generates our BNG S/W switch.

2.2 Programming Protocol-Independent Packet Processors (P4)

Figure 2.2: P4 Abstract Forwarding Model. Source: Adapted from (P *et al.*, 2013).

P4 is a high-level language for programming protocol-independent packet processors that define how the pipeline of a network forwarding device should process the packets using the abstract forwarding model (See figure 2.2). Its specification is maintained by the p4.org [1] consortium and recently joined the Open Networking Foundation (ONF) and the Linux Foundation [2]. Currently, P4 code has a large community involving 60 organizations from the academic and industrial sectors. Thus, P4 language tries to resolve the old limitations of other SDN protocols like OF with which it is necessary to update the version

[1] http://p4.org.
[2] https://p4.org/p4/p4-joins-onf-and-lf.html

each time a packet header is added, maintaining retro-compatibility with previous OF versions. Thus, the support for new protocols is very slow.
P4 language can define the header structures and use the parser to extract the header fields. The pipeline is determined through a set of match-action tables, which execute one or more actions such as packet forwarding, drop and so on. This tables can be filled and modified at "runtime" through the controller. The De-parser writes the packet header fields to send back to the output port. The three main advantages of P4 are:

1. Reconfigurability in the field: Programmers should be able to change how the switch processs the packets once deployed.

2. Protocol independence: Capability to deploy any protocol in a switch.

3. Target independence: To describe packet-processing functionality independent of the H/W where it has been deployed (P *et al.*, 2013).

One of the objectives of this work is to bring support to the core of MACSAD to compile P4 programs of the version 16 . In the next section, we will present the goals of this version 16 and introduce the main elements of the P4 language.

2.2.1 *P*4_16 features

According to (BUDIU; DODD, 2017) the goals of this version 16 are described as follows:

1. Multiple targets support: The previous *P*4_14 versions brought a fixed and rigid architecture to represent a switch with *Ingress* and *Egress* P4 methods. Therefore, the latest version allows defined customized architectures based on the targets (i.e., FPGA, NICs, Software switch, etc.)

2. Simplicity: Many functions and actions were compacted into a simple expression (i.e., the command `add_to_field(a,b)`and was simplified in version 16 to `a=a+b`)

3. Compatibility: Future versions will be retro-compatible with version 16 ; thus the current programs would be compatible to future releases.

4. Extensibility: Some *P*4_14 functions that are not used in all the targets were added to the P4 core language as functions despite these characteristics belonging to a specific target. These functions were converted into objects and *extern* functions as part of Target-specific libraries.

The version 16 syntax is based on familiar C-like language which runs in sequential form, including libraries, and use declarations.

The core abstractions provided by the P4 language can be divided into some categories regarding the pipeline programmability as follows:

- Headers: Fields of the packet header which are analyzed by the Parser. i.e., an Ethernet header syntax declaration could be:

```
header Ethernet_h {
   bit<48> dstAddr;
   bit<48> srcAddr;
   bit<16> etherType;
}
```

- Tables: Logical data structure to perform a type of lookup using an input key to match with the header field. It must define the actions and dimensions of each table.
- Match-Action: Abstraction to execute an action on the packet header field or meta-data.
- Control: Elements and structures to control the data flow including the sequence of tables execution.
- Parser: A state machine with one start state and two final states depending on if the packet parsing is successful (accept) or not (reject).
- Data types: A set of data types to build other data types more complex such as arrays, headers, and structures. A header structure can define its fields with different data types such as bit, int, varbit.
- Expressions: The basic operations are defined in (CONSORTIUM, 2017). It defines a set of operators on Booleans expressions, conditional operators and different operators on data types which are over header fields and meta-data.

2.2.2 *P*4_16 architecture

One of the most important updates of *P*4_16 is to allow us to define the architecture of a switch depending on the target (i.e., a software switch, ASIC, NIC, FPGA, etc.). We need to add the architecture library at the beginning of the P4 program. The creation of the architecture is the responsibility of network devices manufacturers.
The P4 programs add both the standard P4 core library named "core.p4" and the architecture library. In this work we are using the "v1_model" architecture [3] which is the description in *P*4_16 of the old fixed architecture of *P*4_14. This referenced architecture is used in two targets on which the BNG is compiled in this work: on Behavioral-Model 2 the *Simple Switch* target (BMV2, 2018) which is the reference compiler implementation of P4.org and also on the MACSAD compiler. In section 4.1 we have described the testbed to work with *Simple Switch* target with *P*4_16 for the functional verification.
The "v1_model" architecture defines the dataplane for six main functions or controls: Parser, Verify Checksum, Ingress, Egress, Compute Checksum, and Deparser. These functions are used in the design of the BNG dataplane. In the next sub-section, we introduce a P4 compiler and its main compiler parameters to generate different Back-end files.

[3] https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4

2.2.3 *P*4_16 compiler

P4C is an Apache 2 Open Source compiler (P4C, 2018) written in C+11. The compiler data-flow is shown in Figure 2.3. It works on both old P4_14 and new *P*4_16 versions. P4C brings support for the following *back-ends*:

- **p4c-bm2-ss:** Used in the BMv2 *Simple Switch* target. It generates a JavaScript Object Notation (JSON) file.
- **p4c-ebpf:** Used in the eBNF (extended Berkeley Packet Filter). It generates C code which can be run in the Linux kernel;
- **p4test:** Used for learning, testing or debugging purposes. It converts from *P*4_14 to *P*4_16;
- **p4c-graphs:** It generates visual representations of a P4 program;

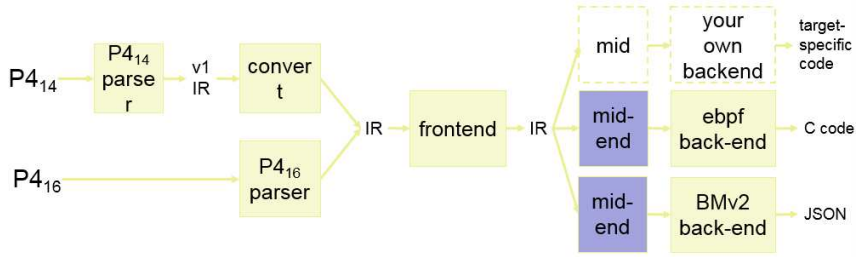


Figure 2.3: *P*4_16 reference compiler dataflow. Source:(BUDIU; DODD, 2017)

2.3 Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD)

This is an Open Source tool which brings an environment to compile and automatically deploy different S/W network devices. It generates the dataplane code for heterogeneous targets bounded to either virtual or physical network interface cards. MACSAD brings high performance through packet accelerators such as Netmap and Data Plane Development Kit (DPDK).
The MACSAD architecture is designed according to three main modules as shown in Figure 2.4a).

- Auxiliary Front-end: The P4 program is the dataplane model representation which is the front-end code. First the P4 code is converted to a High-Level Intermediate Representation (HLIR) using the p4-hlir/HLIR16 modules which support different Domain Specific Language. This IR structure will be used in the MACSAD core compiler to generate C code.
- Auxiliary Backend: This module provides a common Software Development Kit (SDK) for the Compiler by using ODP APIs which bring support for different PktIO frameworks,

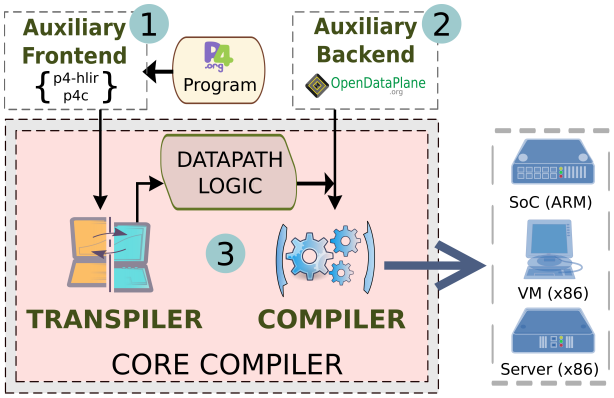


Figure 2.4: MACSAD Architecture & Use Case Workflow. Source:(PATRA *et al.*, 2018)

	Target Independent	Target dependent
Primitive Actions(P4)	Add header, copy header, generate digest, modify field	Push, pop, count, meter
Pipeline Actions	Table Configuration, Protocol Independent Header Parsing	Pkt Rx/Tx, Header Parsing Modify Header Field, Table Creation, Table Lookup

Table 2.1: Packet processing functions. Source (PATRA *et al.*, 2017)

controller interfaces such as Switch Abstraction Interface (SAI), OpenFlow (OF), and some useful primitives actions

- **Core Compiler:** Composed of "Transpiler" and "Compiler" sub-modules (see the figure 2.4b), it transforms the IR generated by the Front-end into the target image in association with the auxiliary Back-end.

The "Transpiler" takes the HLIR input and auto-generates the Datapath Logic codes. Some critical decisions are taken in this sub-module like to optimize the "Dead Code" by identifying reachability in a graph, to decide the type of look-up mechanism to be used, and the size and type of tables to be created.

The "Compiler" module compiles the C-based code from the HLIR representation and ODP hardware abstraction libraries (HAL) to be used in the desired target (x86, x86+DPDK, x86+Netmap, ARM-SoC).

The dataplane functions are divided into two: Target independent and target dependent as shown in Table 2.1. Target-Dependent functions are those that belong to a specific target, e.g., on a Linux system. Target-independent functions are the P4 actions and functions which need to be mapped to ODP APIs such as primitive actions.

2.4 Open Data Plane (ODP)

Open Data Plane projects surged in 2013 with a group of networking SoC vendors which were organized through the nonprofit Linaro Networking Group (LNG). This is an Open Source S/W engineering organization to create an application programming environment for the data plane implementations with high performance and portable across many of H/W platforms. The aim of ODP is to split the application design from the functional implementation, because generally the data plane requires that the application

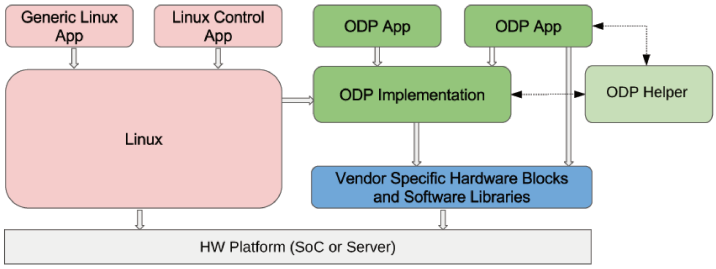


Figure 2.5: ODP system architecture. Source:(ODP, 2018)

must be re-designed on changes in the network speed or capacity because the applications need to be very integrated with specialized H/W to achieve acceptable performance levels. The ODP environment relies on the Linux kernel itself (See Figure 2.5); it consists of a set of ODP common APIs, services, configuration files and utilities which brings an optimized implementation for the underlying H/W.
ODP also bring NIC support by using Vendor Specific HAL (As shown in Figure 2.5) working along with the ODP libraries. Vendors like Intel-DPDK supply much more functionality and performance than a plain NIC can provide. Currently, ODP works with two SDKs for high-speed packet I/O (PktIO): DPDK and Netmap which brings optimization for NICs. However ODP also works with traditional NIC supported in the Linux kernel (Socket-mmap driver).

2.4.1 Applications

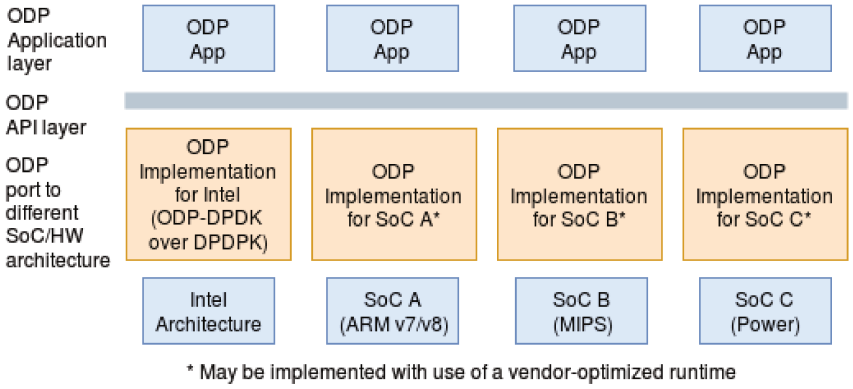


Figure 2.6: ODP Cross-platform implementation view. Source:(MONKMAN, 2018)

ODP applications are organized in threads which can run in parallel with full Linux user processes that implement control and management functions as these processes typically do not have critical performance and latency requirements. Therefore, this tool calls the SDK and optimizes the features and functionalities of Linux-based networking application across many architectures (ARM, Intel, MIPS, and PowerPC). Thus ODP can be easily compiled in ARM-based processors such as Banana Pi, Raspberry Pi and Cavium ThunderX (It has 48 processors and high-perf buses with 40GbE interfaces). Figure 2.6 shows the separation between the application layer and the implementation layer. There are several

ODP applications examples to build a data plane in the ODP Github repository[4] i.e., L2 forwarding (See a C-based code example in the Annex. E.2), l3 forwarding and more which are written in C programming language. This ODP application is optimized to control related SoC/Server resources such as:

- CPUs and memory
- Huge page mappings (how many, what sizes)
- Physical and virtual ports/interfaces
- Packet classification rules
- Scheduler (core groups, algorithms, ordering)
- Hardware queues
- Output traffic management

2.5 Packet generators

2.5.1 Open Source Network Tester (OSNT)

OSNT is an Open Source H/W and S/W platform for network testing. It has been developed in collaboration with Cambridge, Princeton, Centre National de la Recherche Scientifique (CNRS) universities and Google since 2014. OSNT works on the top of the NetFPGA platform. It is available for NetFPGA-10G and NetFPGA-SUME cards which enable flexible network testing and support full line rate through their four 10G Ethernet ports. For this work the NetFPGA-SUME card was used donated by the NetFPGA project.
OSNT brings an open and accurate testing platform that aims to fill the industry packet generators gap which is commonly closed with proprietary solutions and limited flexibility.

OSNT adds 64 bits for TX and RX timestamp per packet. The time-stamp accuracy is because the H/W module controls clock drift and phase coordination controlled by a GPS input.

The OSNT H/W component uses the NetFPGA-SUME card connected to the HOST PC through a PCI Express Gen 2/3 slot. The NetFPGA-SUME must be programmed with a bit file which contains two main H/W sub-systems:

- The traffic-generator which provides PCAP replay function can be shaped to add per-packet delay time and a high-resolution time-stamp (6.4 nsec) used for latency measurements on the Device Under Test (DUT) or Network Function(BNG). Also, the time-stamp mechanism is located just before the transmit 10GbE MAC module.

[4]https://github.com/Linaro/odp/tree/master/example

– The traffic-monitor provides a high-resolution inbound time-stamping, packet filter, packet cutting and hashing in H/W, all with both graphical and command line interfaces.

The main H/W sub-modules to generate the bit file are summarized in Figure 2.7. The OSNT S/W component is composed of the Direct Memory Access (DMA) driver which brings support to send the traffic from the computer to the Packet generator block on the NetFPGA-SUME and userspace software to start the measurements by using a Python script. All the S/W and H/W components are available in the OSNT Github repository in (PLATFORMS, 2018b).

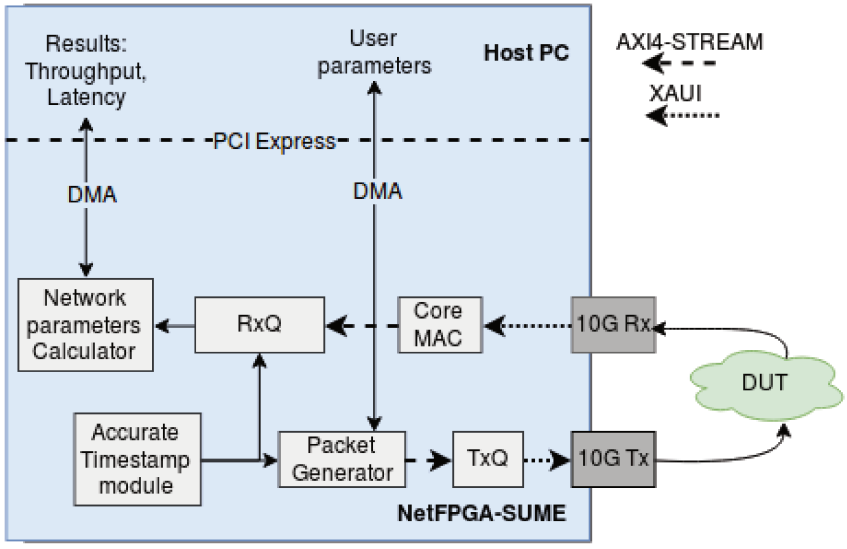


Figure 2.7: OSNT H/W Architecture. Adapted from (ANTICHI *et al.*, 2014)

2.5.1.1 NetFPGA

This an open FPGA platform for research with multiple development projects including OSNT as mentioned before. Currently, there are three supported FPGAs: NetFPGA-SUME, NetFPGA-1G-CML, NetFPGA-10G and other deprecated NetFPGA-1G. Academic institutes can acquire this FPGA platform and licenses as a donation as in our case. This can be done on the NetFPGA website (PROJECT, 2018). In our project the NetFPGA-SUME platform was used which is a Xilinx Virtex-7 XC7V690T FPGA for high-performance networking design. It provides access from data path to the host through 8 lanes PCI-E (Gen3 x8). Furthermore, To implement the OSNT-SUME project we had to meet all the system requirement described in (PLATFORMS, 2018a). Therefore, we spent months meeting them, awaiting Xilinx licenses and implementing our network scenario. We can highlight two 10GE platforms that can be used for the OSNT project; in table 2.2 we can see the platform comparison between NetFPGA SUME and NetFPGA boards.

Feature	NetFPGA SUME	NetFPGA 10G
Approximate Cost	US$ 7000	US$ 3500
FPGA	Virtex 7 690T-3	Virtex 5 TX240T
RAM	8 GB DDR3 SoDIMM 1800MT/s	288 MB RLDRAM - II 800MT/s
SFP+ Ethernet Ports	4 x 10Gbps	4 x 10Gbps
PCI Express	x8 PCIE Gen. 3	x8 PCIE Gen. 1
Power consumption	150W	120W
Timestamp resolution	6.5 ns	6.5 ns
Design time Engineer	months	months
Maximum rate supported Measure	10Gbps	10Gbps
Measure throughput 10Gbps	ok	ok

Table 2.2: NetFPGA Board Comparison.

2.5.2 Network Function Performance Analyzer (NFPA)

NFPA (CSIKOR *et al.*, 2015) is a benchmarking tool to measure throughput (Unlike OSNT that measures both throughput and latency) of any S/W and H/W network function. Its features enable the possibility of evaluating H/W accelerators e.g, programmable network devices, Intel DPDK (INTEL, 2018), Netmap (RIZZO, 2012), OpenDataPlane (ODP, 2018) frameworks and then find the network bottlenecks.

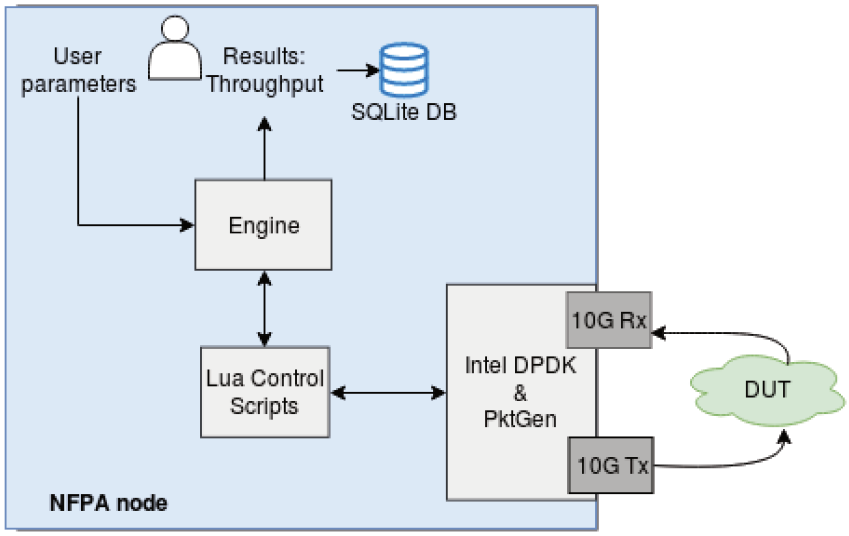


Figure 2.8: NFPA Architecture. Adapted from (CSIKOR *et al.*, 2015)

NFPA relies on the standard libraries of Python. The NFPA network interfaces are built on the recent stable version of DPDK (v17.08) for fast packet handling. The NFPA engine is built on PktGen[5] (v3.4.5) on which the measurements are automated and controlled with Lua scripts. NFPA provides a wide selection of synthetic traces for protocols like Ethernet, IPv4, VxLAN among others, but also is capable of using any PCAP trace. NFPA

[5]1http://dpdk.org/browse/apps/pktgen-dpdk/refs/

supports the following packet sizes: 64, 128, 256, 512, 1024, 1280, 1518 bytes. Figure 2.8 shows that as NFPA sends the packets on Tx port towards the DUT and receives the processed packets on the RX port, it calculates throughput in both Packet/s and bits/s. The obtained results are processed, plotted, and also the measurement data is saved on a database.

2.6 Related work

The work in (ROBERTO *et al.*, 2013) presents a BRAS/BNG implementation using an alternative framework called Click which lends itself to an excellent platform for NFV. It comes with hundreds of tiny simple Virtual Machines (VMs) that can boot quickly and have a small memory footprint. The results reported are related to the overall performance of a number of session establishments and memory consumption when establishing sessions, Also in (DIETZ *et al.*, 2015) they present the impact of design decisions on performance achieving line-rate with medium and higher packet sizes. This is an excellent candidate for NFV adoption.

The work in (MASUTANI *et al.*, 2014) describes a BNG implementation which uses OVS with DPDK PktIO. They present the requirements and basic design of a flexible and elastic network service infrastructure with NFV and SDN by using Open vSwitch (OVS) (PFAFF *et al.*, 2015) which provides performance and stability required by its deployment in ready environments. Unlike our work, they do not present performance results, and their implementation is a protocol-dependent solution.

Recently the work in PISCES (M. Shahbaz et al, 2016) has modified OVS to be usable in conjunction with a P4 (P4_14 version) program. However due to the OVS pipeline construction data plane, reconfigurability is limited.

The work in (NEMETH *et al.*, 2015) exposes the performance implications related to bandwidth constraints and Virtual Network Function (VNF) placement during service deployment. They illustrate this with a "proof of concept" implementation of a BRAS using DPDK (INTEL, 2018). The performance focuses on the QoS function, the most resource demanding function of their prototype, and the impact of core layout and memory usage on performance.

In order to become more flexible (PONGRáCZ *et al.*, 2013) describes the implementation of an L3 forwarding application using the DPDK framework. He reported forwarding rates between 5.26 Gbps and 9.6 Gbps for packet sizes of 64B and 512B, respectively, with one OpenFlow 1.3 implementation as the primary model of SDN data plane. Our routing functions are substantially similar to theirs, but we implement ours in P4 as front-end as well as ODP-DPDK and ODP-Netmap as Back-end.

The work in (HWANG *et al.*, 2015) describes the design of the software router, with

some network functionalities such as Layer 3 forwarding, a firewall that resides in distinct VMs on the NetVM platform obtaining throughputs up to 10 Gbps. Instead of running into H/W limitations such as NIC, their implementation is limited by the available processing capacity.

The work in (VOROS *et al.*, 2018) presents a multi-target high-performance P4 compiler (P4_14 version). In this work, they present a hardware abstraction layer (NetHAL) to bring multi-target support. As a proof-of-concept, they work with two back-ends: DPDK on Intel x86 and ODP on an ARM-based Freescale board. Unlike ours, they implemented L2, L3 and simple Load-balancing use cases and compared them with results in OVS with DPDK over 100GbE . The abstraction level of their NetHAL is higher than ODP in MACSAD; however, it could impact in the dataplane optimization and performance of the generated S/W switch.

In (RAHIMI *et al.*, 2016) they evaluated the impact of specific parameters and settings in a software switch called Lagopus on which were implemented L2 and L3 network layer functionalities using DPDK and OF.
This work focused on evaluating the performance using a software called "Pktgen" achieving maximum throughput of 9.8 Gbps with 1500B of packet size and 100K OF entry tables. On the other hand they studied packet drop rates and presented the impact of parallelization on switch performance, which includes both packet forwarding rate and packet drop rate, at high loads in a configuration with high link rates. This demonstrates the importance of receive-thread packet classification for load balancing and to send delay-sensitive flows to a different worker thread from high-throughput flows.

NetASM (SHAHBAZ; FEAMSTER, 2015) is an intermediate representation that enables for the compiler a high-level and optimized packet processing program for a diversity of targets. They provides a set of Assembler instructions built from the dataplane functionalities. However, it can't be used for high-level solutions (e.g., P4 or OpenFlow).

Main related projects around BNG software switch implementations and data plane support mentioned above are summarized in Table 2.3:

Related works	Application	Target	Domain Specific language support	Platform	Remarks
Rethinking Access Networks with High- Performance Virtual Software BRASes	Click: BRAS	General-Purpose Processor/ Server	No	Open flow, KVM platform	Mostly used for research.
Requirements and Design of Flexible NFV Network Infrastructure Leveraging SDN/OF	OVS: BRAS	Limited by DPDK	Limited	OpenFlow	Optimized for Intel.
The Limits of Architectural Abstraction in Network Function Virtualization	DPDK: BRAS	Limited by DPDK	No	DPDK kit	Intel Data Plane Performance Demonstrators: C language.
Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK	L2	Limited by DPDK	No	OpenFlow	Intel Data Plane Performance Demonstrators: C language.
NetVM: High Performance and Flexible Networking using Virtualization on Commodity Platforms	L3	General-Purpose Processor/ Server	No	KVM platform, DPDK kit	Intel Data Plane Performance Demonstrators: C language.
T4P4S:A Target independent Compiler for Protocol-independent Packet Processors	L2,L3	Limited by DPDK	Yes	DPDK kit	Intel Data Plane Performance Demonstrators: C language.
A high-performance OpenFlow software switch	Lagopus S/W switch	Limited by DPDK	No	Open flow, DPDK kit	Intel Data Plane Performance Demonstrators: C language.
Our BNG S/W switch	BNG	Multi-Target	Yes	MACSAD, ODP-DPDK kit	Currently X86 & SoC (ARMv8) support available

Table 2.3: Feature comparison list of different software switch implementations.

Chapter 3

BNG Switch Design in P4_16

This chapter covers the BNG switch program development using a protocol-independent P4 programming language and the MACSAD compiler. We have divided this chapter into three sub-sections. In the first section, we present the BNG Pipeline coded in P4 language and introduce our dataplane design describing the complexity and principal components. Thus, we describe two action points to give support to the BNG dataplane on the MACSAD compiler, namely, upgrading to the P4_16 version and mapping the P4 functions of the BNG dataplane, such as SetValid/SetInvalid and Drop, into ODP APIs. Finally, we describe our BNG controller's main design features and its role in the network.

Figure 3.1 presents the high-level BNG switch architecture composed of the following three main components:

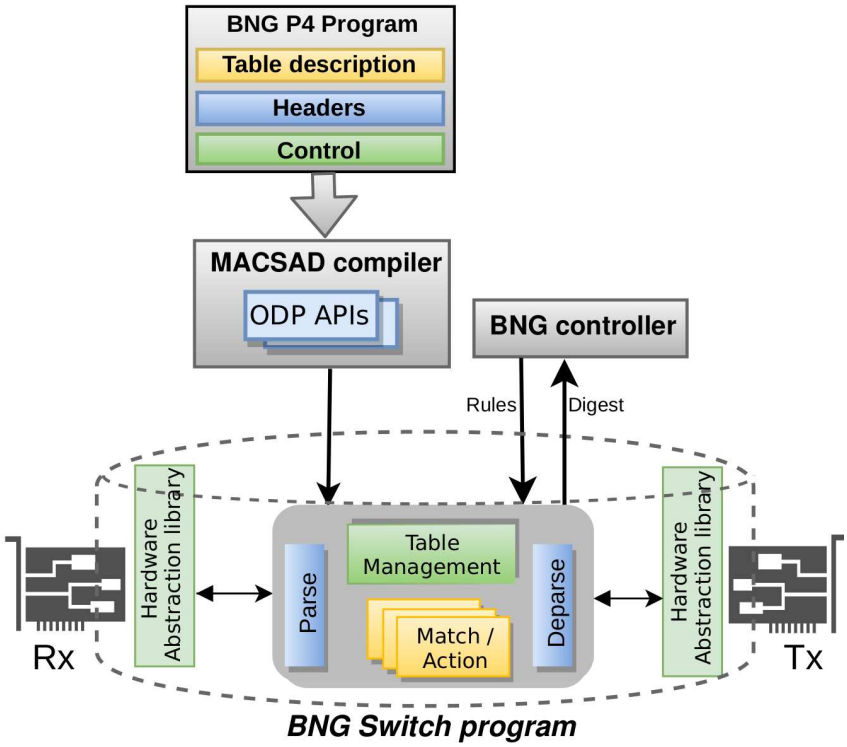


Figure 3.1: BNG switch architecture.

1. BNG P4 program. It gives the BNG dataplane functionalities such as actions, routing

tables, Parsing/Deparsing, and control flow.

2. MACSAD compiler. Which adds support and mapping of the P4_16 program on MACSAD compiler with OPD APIs.

3. BNG controller. Which controls the packet forwarding by populating the routing tables.

Hereby, we describe the implementations and changes made in the three sub-blocks mentioned above.

3.1 BNG Pipeline in P4

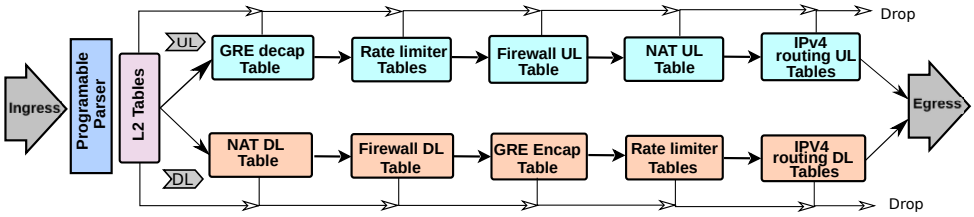


Figure 3.2: BNG data plane.

The P4 program defines all the dataplane functions, such as match/actions, look-up tables, and parser, as well as control flow methods to specify the packet processing flow. The BNG processes five different types of packet headers such as TCP, IP, GRE, ARP and Ethernet, and processes 46 packet fields (See table 3.1).
The dataplane is split into Upload Link (UL) and Download Link (DL) as shown in Figure 3.2 to support the BNG network traffic, i.e., encapsulation when a packet ingresses from the Internet, and decapsulation when the packet ingresses from the CPE. The packet control flow is defined in P4 either in the Ingress and Egress methods.
For a brief impression, the set of look-up tables of the BNG dataplane is shown in Figure 3.2. However the entire graph of the ingress process is found in Figure B.2 in Annex B, and the BNG P4 code of the dataplane is in Annex D as are the other controls and methods defined in the "v1_model" architecture. The graphs of the methods can be generated from the P4 program using the P4C compiler (Mentioned above in section 2.2.3).
The dataplane complexity is summarized in table 3.1, which quantifies and compares it with L2, L3 and NAT use cases complexities developed in previous works. This set of tables and the Parser blocks are described in detail in the next sub-sections.

3.1.1 L2 tables

These define a couple of tables to process Ethernet and ARP packets to establish communication between adjacent nodes at the data link layer level. The L2 protocols allows the BNG to discover the network devices connected to it and save them by performing MAC learning. Therefore, when an Ethernet packet is received from the CPE, an entry is created (or updated) in the L2 table swapping the source and destination MAC address in

		L2	**L3 (IPv4)**	**NAT**	**BNG**
	P4 Version	P4_16	P4_16	P4_16	P4_16
Parsing	#Packet headers	1	2	3	5
	#Packet fields	3	13	26	46
	#Branches in parse graph	1	2	3	5
Processing	#Tables (no dep)	2	2	6	11
	Depth of pipeline	2	2	5	7
	Checksum on/off	off	off	off	off
State Accesses	#Writes to different register	0	0	0	0
	#Writes to same register	0	0	0	0
	#Reads to different register	0	0	0	0
	#Reads to same register	0	0	0	0
Packet Modification	#Header adds	0	0	0	4
	#Header removes	0	0	0	4
Metadata	#Metadatas	1	1	1	11
Action complexity	#Field writes	2	4	6	24
	#Arithmetic expressions	0	0	0	0
	#Boolean expressions	0	0	0	0
Lookups	#Hash_lookups [key_length(bits)]	2 [48]	1 [9]	2 [9], 1[48], 1[16]	2[9],2[48], 3[32], 1[16], 1[8]
	#LPM [key_length(bits)]	0	1 [32]	2 [32]	2 [32]

Table 3.1: P4 use case complexity.

the Ethernet header.
Furthermore, L2 tables help to separate the UL and DL traffic by configuring the BNG network interfaces either as external or internal.

3.1.2 NAT tables

This NAT implementation handles internal and external network TCP traffic. It maps the internal IPv4 Addresses and TCP Ports from the packet headers originating from a client on a private network into an external Port/Address and vice-versa. The TCP packets whose header fields match in the NAT table are processed, rewritten and sent forward appropriately or when a TCP packet is received on the internal/external interface, and if its header fields are not mapped in the NAT table, then the packet is dropped.
The P4 code of this table was re-used to create a different P4 NAT use cases whose complexity is shown in Table 3.1. Also, the results of NAT use case are published in (MEJIA *et al.*, 2018) as part of the publications released during this work. All the publications are referenced in the Annex A.

3.1.3 GRE tables

For DL traffic in the internal network, the BNG creates a point-to-point tunnel by encapsulating the packets with the GRE packet header standard (FARINACCI *et al.*, 2000) and saves the user ID in order to establish the user session.
In the reverse direction (UL), CPE-originating packets are decapsulated to forward to the external network.
The *SetValid* (For packet encapsulation) and *SetInvalid* (For packet de-encapsulation) are examples of functions implemented in the Auxiliary Backend using ODP APIs which are explained in detail below in section 3.2.

3.1.4 IPv4 tables

The routing table stores the next hop information (IP address and output port) based on the IP destination address. It's based on the Longest Prefix Match (LPM) implementation. For L3 forwarding, the switch must perform the following actions for every packet: (i) update the source and destination MAC addresses, (ii) decrement the time-to-live (TTL) in the IP header, and (iii) forward the packet to the appropriate output port.

3.1.5 Parser

The BNG Parser will detect the different packet headers to perform the match/action functions. It extracts header fields from the packet at the current offset into per-packet header instances and marks those instances valid, updating the parsed representation of the packet. The parser then indicates as valid byte the correct header and makes a state transition to the next header. Only headers with valid packet headers are serialized, in our case, the output packet is serialized with the BNG UL/DL packet headers. The parse graph is shown in the Figure 3.3.

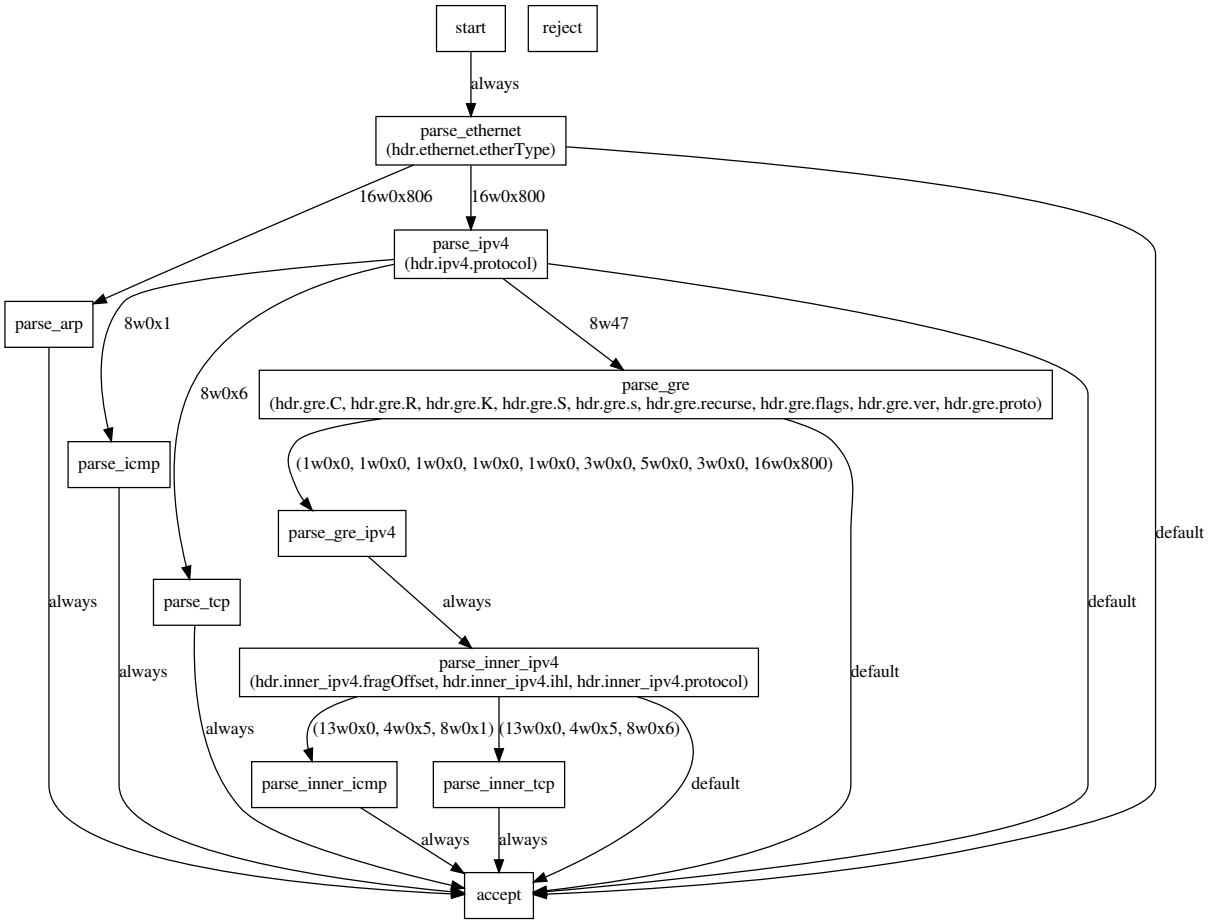


Figure 3.3: Parser graph from P4 BNG program.

3.1.6 Rate limiter

The work in (POPA *et al.*, 2013) and (BALLANI *et al.*, 2011)] presents some solutions to host-based rate enforcement for guaranteeing bandwidth allocation in a multi-tenant data-center. The authors show the limitation of legacy rate limiter implementations in a data-center which requires a large number of rate limiters, far from the capacity of commodity NICs (e.g., Intel's 82599 10G NIC supports up to 128 rate limiters).
In our BNG implementation the Rate limiter is defined in P4 code by using two tables in which one rate limiter can be configured per subscriber without slowing down performance. This rate limiter implementation is based on (HEINANEN; GUERIN, 1999) (A Two Rate Three Color Marker) RFC standard which describes traffic policy elements like a meter and a dropper. The meter measures the traffic and determines whether it exceeds the rate limit, and when it occurs the packets could be dropped.

3.1.7 Firewall

A firewall is a key device by preventing the network from outside attacks. The difference between a typical firewall and ours is that its policies are applied in the whole network instead of the filter at the boundaries. Therefore, the internal traffic can be seen and filtered through the separation of data plane and control plane for adding DROP actions,

and rules onto the switch tables.
Our BNG implementation approach consists of pre-loading a set of rules onto the switch tables, which can be added using a text file filled with dozens of thousands of entries for IP destination addresses.
The SDN controller for this table is coded in C language in order to add the entries each time that BNG is started. The command to add this rule at CLI is the following:

```
> f 10.0.0.10
```

Where "f" is the command to filter an IP destination address within the table entries file.

3.2 MACSAD Support of P4_16

MACSAD is in charge of compiling a P4 program into C code. Since its creation, MACSAD brings support to P4_14 (v1.0/v1.1) versions. However, P4_16 new version was launched in May 2017, bringing some changes regarding language syntax and semantics. The P4 evolution (see Figure 3.4) shows how it has been transformed from a complex language (more than 70 keywords) into a stable and optimized instructions set (less than 40 keywords) accompanied by a library of fundamental constructs that are needed for writing most of the P4 programs. Therefore as part of this work we give the MACSAD support to both P4_14 and P4_16 versions by replacing the HLIR plug-in in the Auxiliary Front-end of MACSAD with a new one. The new plug-in and P4_16 support will be explained below.

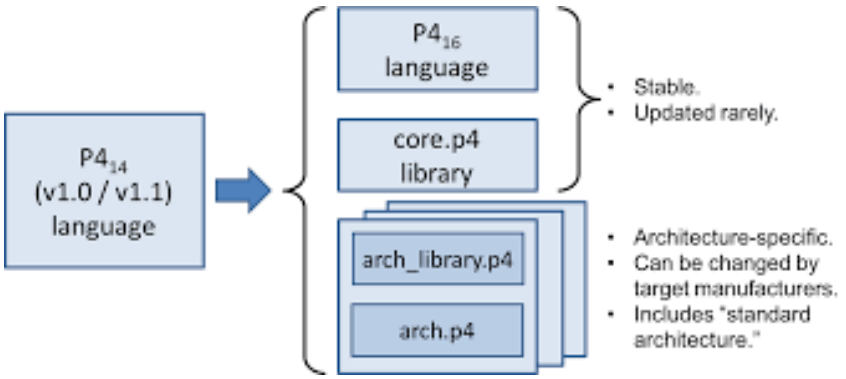


Figure 3.4: Evolution of the language between versions P4_14 (v 1.0 and 1.1) and P4_16.

3.2.1 Intermediate Representation for P4_16

The HLIR is a language abstraction and target-independent which contains enough information about the dataplane produced in front-end code.
The HLIR for P4_16 module (ELTE, 2017) is maintained by the P4 group in Eotvos Lorand University (ELTE) at Budapest-Hungary which brings an IR from the P4 code. It uses the P4C compiler to generate a JSON file from a .p4 source file to create a convenient Python representation of it. We can run HLIR16 in a standalone way using Python 2 or 3 as follow:

```
> python test_hlir16.py "bng.p4" 16
```

The IR is used in MACSAD in the “Transpiler module” which brings access to different P4 objects such as p4_parser, p4_tables, p4_actions, p4_headers and others ordered in a data structure of Python.
The previous version of MACSAD used the p4-hlir project[1] to translate the P4 program (P4_14) into an IR and map the P4 code into ODP APIs. Therefore, in this work, we updated the MACSAD Front-end on the "transpiler" module by replacing the HLIR API by the HLIR16 module. The "transpiler" module calls the HLIR16 libraries and generates a Python representation of the P4 code which is used later to map the P4 functions with ODP APIs in the core compiler on MACSAD. The MACSAD architecture and support for the old P4_14 is still the same.

3.2.2 Integration with OPD API's

The previous MACSAD version gives support for essential dataplane functions like *copy_header* and *generate_digest* but it has no support for other P4 primitives actions such as add/remove the header, drop packet function and other own capabilities of the target device which is necessary for the packet processing of the BNG implementation. During this work, the required P4 functions are appropriately mapped with the ODP APIs in order to support the BNG dataplane.
This P4 function mapped into ODP functions is done over the independent target functions on MACSAD, which uses the HLIR of P4_16 code to generate C-based code automatically. Each P4 function needs to be added as a Python method which generates the C-based code and calls the correspondent ODP function.
In the following sub-sections, we will describe the implemented P4 functions on MACSAD used for the BNG S/W switch.

3.2.2.1 Set Valid/Invalid functions

The BNG implementation uses the packet Encap/De-captulation function using the GRE protocol. Therefore we have added the actions into the MACSAD Packet processing primitive functions as shown in table 2.1. To map the *SetValid/SetInvalid* P4 functions en-charged of packet Encap/De-capsulation in ODP, we need to add a Python routine to compile the functions and later create the target Dependent functions using the ODP APIs.
We have defined two ODP functions: setInvalid/setValid functions which uses the ODP APIs, i.e. push and pop methods to add and remove the packet headers. This code is referenced in the Annex C.

3.2.2.2 Drop Function

This is a key action executed in many tables across the pipeline. If the table lookup fails, the packets could be drooped. For BNG this action is used extensively in the Firewall table.
In P4_16 the *mark_to_drop()* function is also mapped in ODP. We have defined the

[1] https://github.com/p4lang/p4-hlir

mark_to_drop() ODP function which uses a flag to drop the packet. This code is referenced in the Annex C.

3.3 BNG Controller Implementation

The SDN controller is considered the brain of network devices. The forwarding rules are defined at the top of the controller. In many cases, SDN controllers rely on standardized modules. However, our controller is created locally as a proof-of-concept, allowing customized control of network functionality.
The role of the controller is essential as it interacts with the MACSAD BNG switch to control the packet forwarding populating the tables automatically from a trace file and by receiving digest messages to report new source MAC addresses added to the switch. The controller has functions to add, set the entries tables with default values and to fill the BNG tables automatically from a text file. We can add many entries (1-100k table entries) to the switch by using the the BNG controller entirely coded in C programming language as we can see in Annex E.1. The *"messages.h"* library contains the methods and structures to encapsulate and establish communication with the BNG throughout messages. The *send_p4_msg()* function in the controller encapsulates the messages to add the table entries and send to the BNG switch.

3.4 Limitations

The implementation of BNG covers many functionalities as it handles all the access and core network traffic. In this chapter key functionalities are developed using P4 which will be evaluated in the next chapter. However, our implementation can be improved by using other complementary BNG functions or optimizing the current ones added and mapped in the MACSAD compiler. We can describe some limitations of our P4 BNG implementation as follow:

- BNG Controller interfaces. A common SDN controller has two interfaces: On one hand, it has the Southbound interface allowing communication between the higher level components which configure the network application. On the other hand, the Northbound interface use the OpenFlow protocol which enables communication between the controller and Switch. In our implementation, the BNG controller has a single southbound interface instead of both. Therefore, it is necessary to improve the controller interfaces following the SDN standard.

- Lack of QoS support. An ISP must apply QoS policies to his users over the network. There are some mechanisms for achieving these kinds of goals. In the dataplane, it is possible by implementing priority queuing and traffic management blocks, but due to the complexity of these blocks and limited documentation about ODP APIs, this will be performed in future works.

- Rate limiter block is implemented with the P4 primitives by using the "meter" function which is in the v1_model architecture. This function works well in the *simple_switch* target which brings support for almost all the primitives of v1_model architecture. However for MACSAD, this primitive is under construction. Although the tables and related actions are added to the BNG pipeline, there are no entries in the controller that can affect the functionality and performance of MACSAD. We plan to bring support for this in a near future.

3.5 Concluding Remarks

In this chapter, we have presented our design and implementation of the BNG using P4_16 code following the MACSAD workflow which brings programmability, portability, and performance. Also, we summarized the pipeline complexity of BNG and other MACSAD use cases from previous works in table 3.1. Here we can observe that BNG has more than twice the NAT tables and BNG performs much more look-ups than NAT, therefore it is expected that BNG has lower performance compared with NAT. The perfomance results of L2, L3, NAT and BNG will be shown in the next chapter. All the use cases are available to the community in the INTRIG Github repository[2].

The P4 language brings a high-level of programmability to the BNG dataplane, which is a desirable feature to bring more network flexibility and target-independence. That's why P4 programming language has been widely adopted by industry and educational institutions.

HILR16 abstracts the data-plane functionalities such as Parser, tables, controls, and actions to enable high-level data plane representations, providing an efficient assembly of these primitive actions for the MACSAD compiler.

The dataplane abstraction level that HLIR16 gives to the MACSAD compiler allows the facility to add the P4 BNG functions such as SetValid, Set Invalid and Drop functions which can be used in similar dataplane applications.

"Transpiler" block is based in the HILR-16 ELTE module which enables P4_16 language to provide more abstract data plane representation and provides an efficient assembly of these primitives in the MACSAD compiler. This block makes possible the mapping between the BNG dataplane and ODP APIs.

In the next chapter, both the BNG switch program and the controller will be evaluated in a virtual and realistic scenario applying customized network traffic and analysis of the results.

[2]https://github.com/intrig-unicamp/macsad-usecases

Chapter 4

Evaluation

In this chapter, we present the procedure used to evaluate our prototype. We chose two methods to perform this evaluation: The first is focused on assessing the BNG S/W switch functionalities during the entire development process. For fast-prototyping, the switch is built on a P4 S/W Switch, named Behavioral Model 2 *simple_switch*[1], which works with Virtual Ethernet (vEth) interfaces to send/receive the BNG network traffic. In the second method, the BNG S/W switch is built on the MACSAD compiler running over a commodity server in a realistic network testbed in which we evaluate the performance using NFPA (S/W based) and OSNT (H/W based) traffic generators to show the impact of them over the BNG use case. We consider a method to generate the traffic to test the BNG capabilities for a simple (best-case) and complex (worst-case) scenarios. The traffic generator sends and receives the packets from/to the DUT through two 10G SFP+ NICs to collect and analyze the network metrics. Both evaluation methods and results are explained in the next two sections.

4.1 Functional Evaluation

The functional network evaluation scenario is shown in Figure 4.1. To evaluate the BNG functionalities mentioned in subsection 3.1 our evaluation scenario has two main parts, the P4 S/W switch (DUT) and the tester which are defined as follow:

- P4 S/W switch: The BNG dataplane is targeted for the *P4 Simple Switch* in order to run, analyze and debug its main BNG functions. This P4 S/W switch brings a CLI-based controller interface on which we can add the rules according to the tables on the dataplane.
- Tester: The "run_test.py" Python script which is placed in the MACSAD Github repository (MACSAD, 2018) is used to create customized packet flows, configuring Tx and Rx vEths to send the packets through the BNG Upload/Download links. The BNG traffic generation is described in section 4.2.1 using the Scapy Python library.

[1]https://github.com/p4lang/behavioral-model/tree/master/targets/simple_switch

The packet analysis is a complex task in the process of the design and functional validation of P4 S/W switches. We use Wireshark[2] a friendly graphical interface network protocol analyzer which can capture the packets across either virtual and physical network interfaces, which is very useful in the switch development task. The BNG tested functions during the entire development process are:

- Lookup tables
- L3 forwarding
- MAC address learning
- GRE Encap/Decap
- NAT
- Firewall
- Rate Limiter

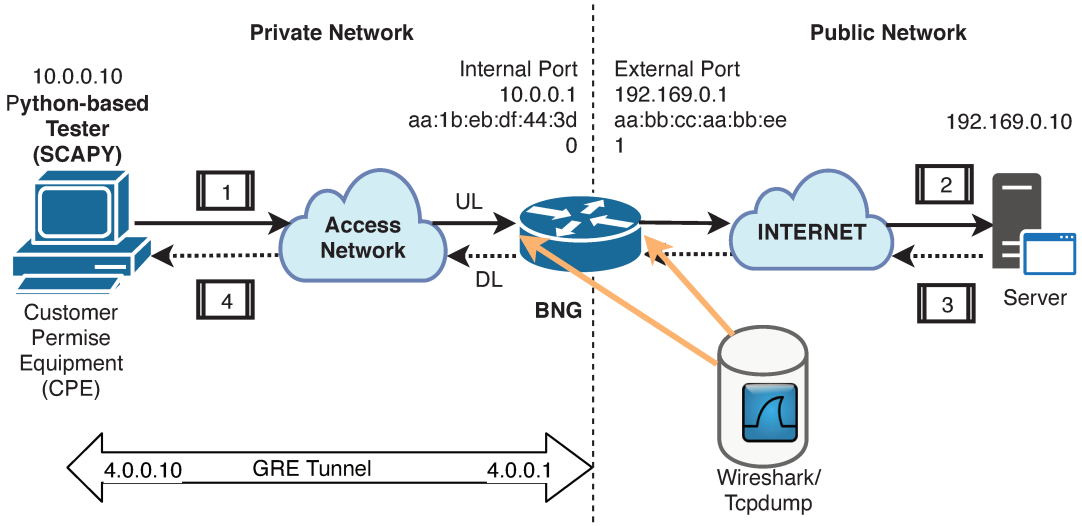


Figure 4.1: Functional Evaluation Scenario

All these functions are evaluated using the scenario from Figure 4.1 which shows the BNG network traffic between the Private and External (Public) networks. The data-path is divided into an Upload Link (UL) coming from the *CPE* to the *Server*, and Download Link (DL) path from the *Server* back to the *CPE* which are explained as follow:

- **UL:** Scapy tool generates the GRE traffic (i.e., encapsulating original TCP/IP) simulating a home gateway wich encapsulates the CPE packets with GRE (See in Figure 4.2 the CPE traffic packet No. 1) and sends the packets through the access network towards the BNG. When the packet arrives at the BNG S/W switch, it performs L2 address learning, verifies user ID, and decapsulates the GRE packet.

[2]https://www.wireshark.org/

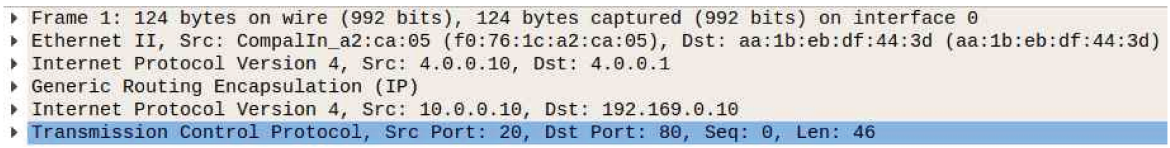
The NAT table rewrites the inner headers where the source TCP/IP headers are rewritten with the public TCP/IP port.
Finally, the BNG performs IPv4 packet forwarding and selects the output port to send the packet towards an external destination server (192.169.0.10)(See in Figure 4.2 packet No. 2).

- **DL:** The *Server* (192.169.0.10) sends TCP/IP traffic back to the CPE (10.0.0.10) via BNG S/W switch through the external interface (See in Figure 4.1 packet No. 3). The BNG performs NAT, adds the point-to-point GRE tunnel header, writes the IPv4 outer header, and completes the IP packet forwarding selecting the next hop and output port towards the CPE (See in Figure 4.2 packet No. 4). performing GRE encapsulation.

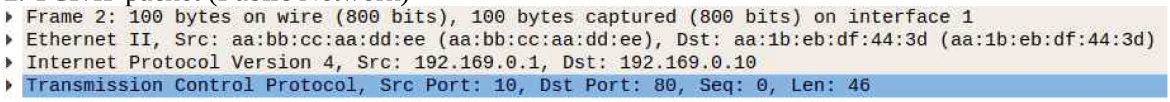
The firewall is tested by adding the corresponding L3 entries tables then all the packet headers that match in the firewall table are dropped.
Also, the rate limiter is tested by dropping the packets whether the traffic exceeds the rate limit according to the policies described in section 3.1.6.
Figure 4.2 corresponds to the packets described above and captured in the vEth interfaces using the Wireshark tool[3].

Upload Link (UL)

1. GRE packet (Private Network)



2. TCP/IP packet (Public Network)



Download Link (DL)

3. TCP/IP packet (Public Network)

4. GRE packet (Private Network)

Figure 4.2: Captured BNG Packets on vEth interfaces

[3]https://www.wireshark.org

4.2 Performance Evaluation

The objective listed in chapter 1 is to evaluate the performance in terms of network metrics like throughput and latency. In this section, we show how the switch performs across different workloads and configurations.
Considering that we could have millions of packets through the BNG switch in a network it is important to asses the performance at high-speed. To validate the BNG S/W switch in a real network environment, we built it with MACSAD compiler running over a commodity server. The BNG interfaces are connected to two different traffic generators placed in different nodes. The first Tester node runs OSNT and the second one runs NFPA packet generator. Both are connected to the BNG MACSAD Server node via two 10GB SFP+ network interfaces.
Measurements are conducted with traces tailored to the BNG UL/DL traffic for the different number of flows (1 to 100K flows). The BNG S/W switch is compiled and evaluated with both DPDK and Netmap PktIO drivers.

4.2.1 Traffic generation

To asses the BNG implementation, there are two methods to generate traffic:

- Generating synthetic traffic: generated packets are sent according to the specification of the DUT varying the workload parameters like packet headers, packet size and data rate.

- Replaying the traffic: The received traffic of a commercial network is saved in a trace file, and then it is replayed in the device under test.

We select the first method to asses the performance of the BNG software switch because we can vary the workloads according to the BNG switch features. We create random traffic for all the experiments to send to the DUT with varying packet sizes and Forwarding Information Base (FIB) sizes (1,100,1k,100k).
The BNG PCAP traces and FIB entries are created in Python using the Scapy libraries since it generates PCAPs with different network protocols such as Ethernet, IPv4, GRE, TCP used in our validation.
The FIB entries are filled with the BNG Controller. The filling time on MACSAD must vary depending on the number of FIB size, i.e., 100 FIB entries spend approx. 20 seconds.
For the BNG best-case test, the NFPA node transmits packets with a fixed source and destination MAC addresses, IP addresses, TCP port, and tunnel IP addresses in order to process the packets faster by matching a single table entry. The configurations of the tables are shown in figure 4.3.
For a complex use case, the NFPA provides a wide selection of traffic traces with different packet headers and sizes, for example with 1 to 100k flows of tables entries and each one having a random destination and source MAC addresses, source and destination IP addresses, TCP ports, GRE IP addresses.

IPv4_lpm table		
Inner IP	Port	MAC
10.0.0.10	0	aa:1b:eb:df:44:3d
192.169.0.1	1	aa:bb:cc:aa:bb:ee

GRE tunnel	
Outter IP Src	Outter IP Dst
4.0.0.10	4.0.0.1

NAT table			
Inner Src IP	Inner Dst IP	TCP Src port	TCP Dst Port
192.169.0.10	10.0.0.10	20	80
10.0.0.10	192.169.0.10	80	20

Figure 4.3: Best-case Tables information.

4.2.2 Test setup

The traffic generation is performed by OSNT and NFPA separately. In NFPA all measurements are conducted for 60 sec as per (BRADNER; MCQUAID, 1999).
The configuration of the servers used in the test-bed from figures 4.4a, 4.4b and 4.6 mentioned in the next sub-section is shown in the following text boxes.
The MACSAD node is configured with ODP (v1.19)[4] and both DPDK (v17.08)[5] and, Netmap (v11.3)[6] PktIOs. In the following text box is shown the H/W Server configuration :

- **Processor**: Intel Xeon D-1518 processor 4x2 cores with 2 threads per core running at 2.20 Ghz.
- **Memory**: 32GB*2 DDR4 SDRAM
- **OS**: Elementary OS 0.4.1 Loki (Linux kernel 4.13.0-32-generic)
- **NIC**: dual-port 10G SFP+

The first tester (Tester 1) performs latency and throughput measurements using OSNT which brings high accuracy on testing through the NetFPGA-SUME card. The configurations are listed in the box below.

- **Processor**: Intel(R) Xeon E5506 processor 4x2 cores at 2.13 Ghz
- **Memory**: 8GB*2 DDR3 SDRAM
- **OS**: System Ubuntu 16.04 (Kernel 4.4) LTS

[4]https://github.com/Linaro/odp/releases/tag/v1.19.0.0
[5]https://github.com/DPDK/dpdk/releases/tag/v17.08
[6]https://github.com/luigirizzo/netmap/releases/tag/v11.3

- **NIC**: dual-port 10G SFP+
- **NetFPGA SUME card**:
  - **FPGA**: Virtex-7
  - **Logical elements**: 693K logical cells x8 Gen. 3
  - **SFP+ interfaces**: 4
  - **On-chip memory**: 51Mbits
  - **DRAM**: 2xDDR3 SoDIMM: 4Gbytes
  - **SRAM**: 27Mbyte QDRII+, 500Mhz
  - **Storage**: MicroSD, 2X SATA 128-Mbyte Flash

The second tester (Tester 2) generates traffic and performs throughput measurements using NFPA. The configurations are listed in the box below.

- **Processor**: Intel Xeon E5-2620v2 processor(4 cores) at 2.20 Ghz.
- **Memory**: 8GB*8 DDR3 SDRAM
- **Operating**: Ubuntu 16 LTS (Kernel 4.4)
- **NIC**: One dual-port 10G SFP+

4.2.3 Throughput

4.2.3.1 Baseline Throughput of the Traffic Generators

This test evaluates the highest throughput that the traffic generators can reach for both OSNT (Figure 4.4a) and NFPA (Figure 4.4b) test-beds. The experiment starts generating Network traffic at Tester one and two independently using the L2 trace file with different packet sizes (98, 128, 256, 512, 1024, 1280, 1580). The generated packets are sent by the port0 (NF0/P0) and received in the port1 (NF3/P1) at the NetFPGA/NIC card respectively.
Figure 4.5 shows that NFPA can achieve line rate for all the packet sizes as long as OSNT achieves line rate except 98 and 128 Bytes packet sizes. It means that the maximum throughput with small packet sizes for the next experiments won't overcome 8.7 Gbps and 10 Gbps (14.8 Mega packets per second (Mpps)) with OSNT and NFPA respectively.

(a) OSNT testbed (b) NFPA testbed

Figure 4.4: Testbed in standalone mode

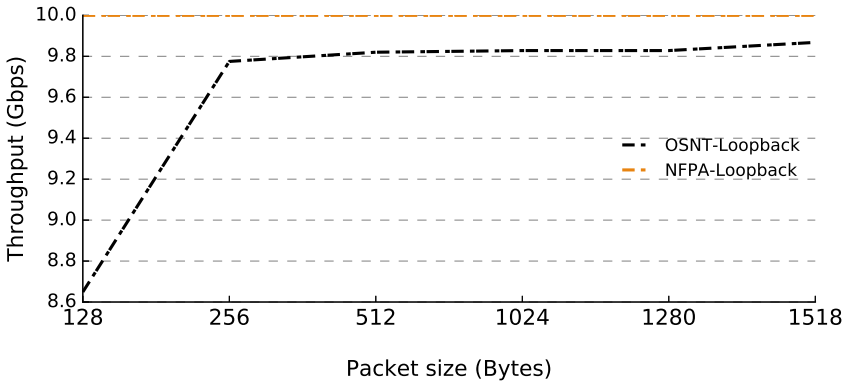


Figure 4.5: Throughput in loop-back mode.

(a) OSNT testbed for latency and Throughput (b) NFPA testbed for Throughput

Figure 4.6: Testbed to evaluate Latency and Throughput in the BNG S/W switch

4.2.3.2 Impact of the Traffic Generator tool for different packet sizes

We have analyzed the performance of BNG for different packet sizes using the two relevant open source S/W and H/W packet generators: NFPA and OSNT respectively configured as in Table 4.1. For this experiment, we have configured two different test-beds: the first one the DUT is connected to OSNT traffic generator as shown in 4.6a, the second one the DUT is connected to NFPA tool as shown in Figure 4.6a.

The switch workload is shown as follow:

- MACSAD is configured with a burst size of 32, two cores with two threads each (two for TX queues and two for RX queues) and DPDK PktIO. We compile and run an instance of the BNG S/W switch connecting two SFP 10+ NICs: Port 1 and Port 2 attached.
- We run the traffic generator with 100 FIB entries and traces to match and forward the packets from Port 1 towards Port 2.

Packet Generator	Version	IO Driver
Pktgen-DPDK (NFPA)	v2.8.4	DPDK 2.2
OSNT	v1.7.0	SUME RIFFA v1.00

Table 4.1: Configuration of Packet generators

- Use a BNG UL/DL traffic flow with different packet sizes (128-1518).

In Figure 4.7 we observe the performance of MACSAD obtained for BNG UL/DL. NFPA achieves higher values than OSNT for all packet sizes. The difference in throughput between NFPA and OSNT for BNG DL with smaller packet sizes (128B,256B) is 20-30 % and 5 % for BNG UP.

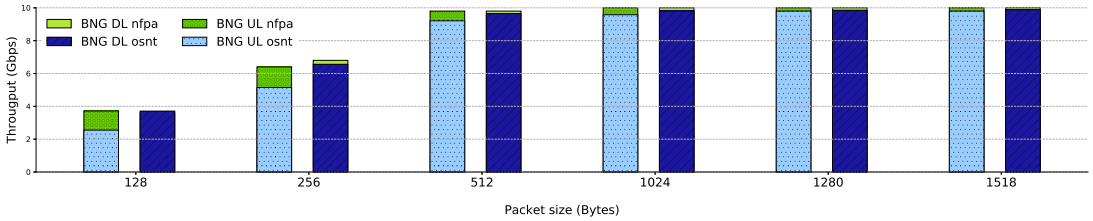


Figure 4.7: Throughput of BNG UL/DL with DPDK PktIO using NFPA and OSNT.

This is explained in (EMMERICH *et al.*, 2017) where the measured error for high rates using Pktgen-DPDK which is the core of NFPA is more than 16 %, and its resolution is 0.195 Mpps. That is why the throughput does not vary gradually with NFPA achieving line rate faster as long as with OSNT the throughput increment slowly. Thus, line rate for BNG-DL and BNG-UL is achieved just for 512-1518 packet sizes with NFPA while line rate with OSNT is achieved from 1024-1518 packet sizes.

4.2.3.3 Impact of FIB and Packet Sizes

For this experiment, the throughput measurements are performed with OSNT traffic generator as it achieves accuracy metrics. We have connected the DUT to OSNT as shown in Figure 4.6a. The throughput performance for different packets sizes (74B/98B-1518) and different FIB sizes (100, 1K, 10K, 100K). MACSAD is configured with 32 of burst size, two cores, and DPDK PktIO.

Figure 4.8 shows that BNG can achieve line rate for medium and bigger packet sizes

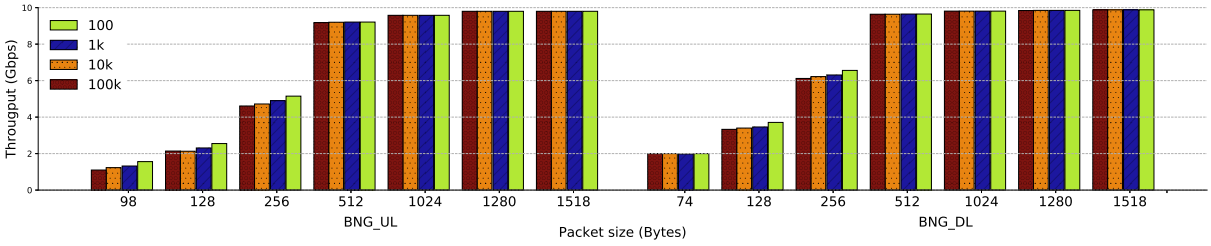


Figure 4.8: Impact of FIB sizes in the Throughput for BNG with DPDK.

(512B-1518B) irrespective of the FIB table size. However, there is still a system bottleneck

for the smaller packet sizes (74B/98B, 128B, and 256B) as it handles more packets per second and the BNG needs to process more headers, tables actions using more hardware and software resources than bigger packet sizes which handle fewer packets per second.

4.2.3.4 Impact of burst Size

In this experiment, MACSAD is configured with two cores, both Netmap, and DPDK PktIOs. The test is performed by using the testbed from Figure 4.6a with OSNT and by adding 100 FIB entries to the switch.

Figure 4.9 shows the results for BNG DL for different burst sizes. The lines in blue

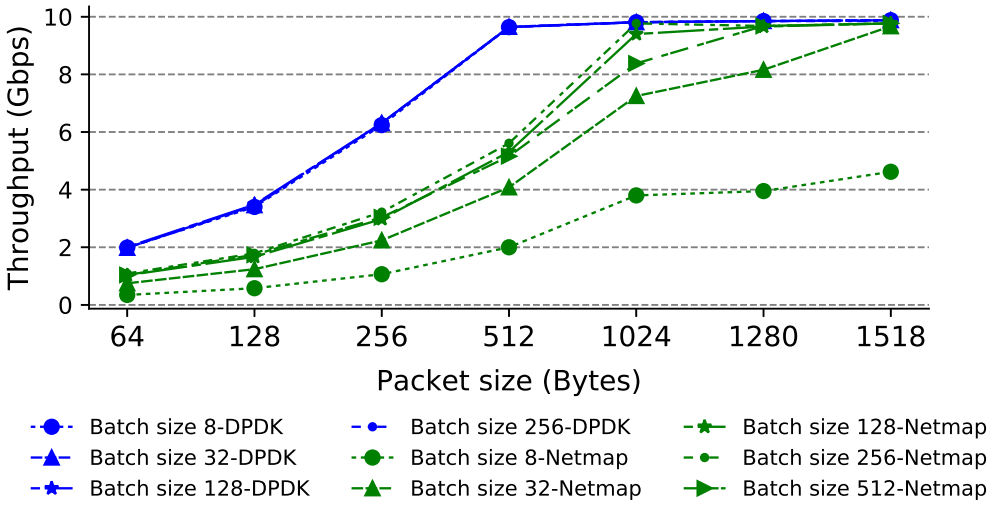


Figure 4.9: Impact of burst size in the Throughput for BNG-DL with DPDK and NETMAP with 1k random FIB entries.

represent the throughput with DPDK PktIO and the lines in green with Netmap PktIO. The packet processing with DPDK is faster, achieving line rate for bigger packet size (512B-1518B). There is not much difference of performance using different configurations of burst size with DPDK mainly because packet processing in DPDK is faster independently of the setting of burst size. However, we achieved a slight increase of performance with a burst size of 32.

The packet processing with Netmap PktIO becomes a bottleneck as it relies on system calls for the packet handling; then the increase of burst size can help to improve the performance in throughput. The best throughput results are with burst sizes of 256.

As expected, for both PktIO accelerators the worst case is for smaller packet sizes. One approach to overcome these performance limitations is employing more CPU cores. Indeed, the works in (MEJIA *et al.*, 2018) and (MEYER *et al.*, 2014) also show how to scale the packet processing by employing more CPU cores.

Figure 4.10 shows the packet processing scaling in BNG_DL with burst size of 256 by using 1,2 and 3 cores. Surprisingly the performance is better using two instead of three CPU cores. Therefore it is necessary to allocate the CPU resources dynamically depending

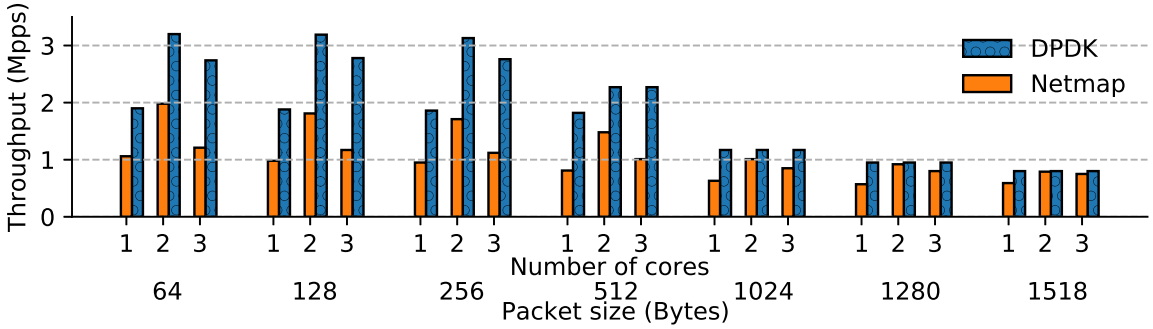


Figure 4.10: Impact in the throughput of the number of CPU cores for different packet sizes on BNG_DL. (100 random flow entries, DPDK and Netmap, 256 burst size).

on system load or other factors (e.g., energy consumption). In (PATRA *et al.*, 2018) a Dynamic adaptive CPU core allocation technique is presented for MACSAD under evaluation which could be applied in BNG for better CPU resources allocation in the future.

4.2.3.5 Impact for increasing the pipeline complexity for different FIB and packet sizes

BNG has a set of tables for processing headers such as Ethernet, IP, TCP, GRE, ARP which increment the pipeline complexity. Either L2, L3 and NAT network functions are presented as independent MACSAD use cases in previous works in (PATRA *et al.*, 2017) and (MEJIA *et al.*, 2018).
In this experiment, we have compiled the L2, L3, NAT, and BNG P4 use cases separately and performed tests independently by using OSNT to generate and capture the network traffic on each of them. For each use case, we used a set of FIB sizes (1, 100, 1k, 100k), smaller packet sizes (64/98B for BNG DL/UL and 64B for L2, L3, and NAT) and medium packet sizes (256B-512B) because for bigger packet sizes the packet processor can achieve line rate. MACSAD is configured with 32 of burst size, two cores, and DPDK PktIO.
Figure 4.11 shows that the performance decreases linearly with the increment of the data

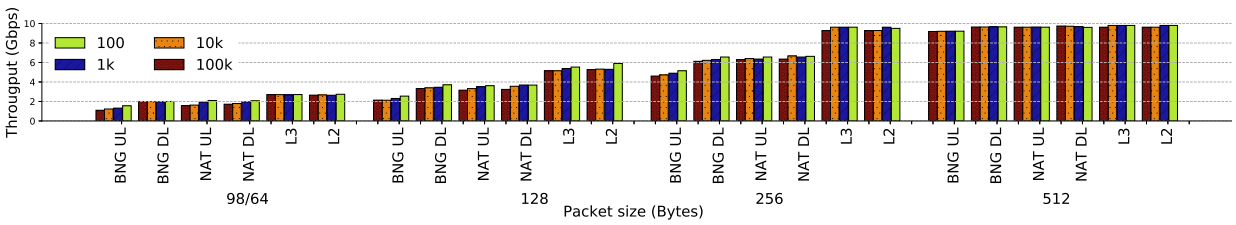


Figure 4.11: Throughput for increasing pipeline complexity, FIB and packet sizes with DPDK.

plane complexity. All the uses cases can achieve line rate for the medium packet and bigger packet sizes from 512. The result proves that the use of some functions such as LPM lookups, hash lookups, Drop, Set Valid, and Set Invalid in complex use cases impact on the performance (, i.e., L2 use case has two hash look-ups while BNG has nine).

4.2.4 Latency

All the packet processing latency tests are performed with OSNT by using the testbed from Figure 4.6a, on which the packets are time stamped in H/W on the Tester 1. The Tx/Rx timestamp values are added in the packet in a pre-configured position. Tx timestamp is attached to the packet immediately before sending and RX timestamp after receiving the packets at the physical layer respectively. OSNT achieves 6.4ns timestamp resolution which increments the latency measurements precision.
Often the Tx/Rx timestamp values should be in the payload part otherwise whether it is timestamped in the header could affect the packet parsing and processing. This process works fine whenever the packets still have the same size (without adding/removing new headers) later than the packet processing.
In this work, we present an approach to solve this issue aiming to achieve latency measurement. We are adding a new "timestamp" header in both the (TCP packet for the Download link or GRE packet for UL) at the beginning of the packet. The timestamp header has two fields: ts_rx, ts_tx with 8 bytes each thus, OSNT script is configured to timestamp the Tx/Rx values at the first part of the packet.
In the BNG P4 program, the parser is updated just for the latency experiments (See the BNG P4 parser graph in the Annex B- Figure B.1) by adding the "timestamp" header. However since there are not new tables linked to this new header there is no impact in the performance, neither affecting the normal function of the switch by parsing in the same way the GRE, TCP, IP, Ethernet headers or by executing the corresponding forwarding tables.
The BNG latency experiments in the next sub-sections are tested by sending 100 sequential packets to the DUT and repeated 10 times. The OSNT manages the traffic rate by configuring the Inter-Packet Gap (IPG). Our approach is to measuring the latency with low latency following others work such as (KAWASHIMA *et al.*, 2017) then, for our next latency experiments we have set with 10% of line rate by configuring the OSNT script with IPG in 100000 in order to bring a stable conditions to measure the latency experiments. Measurements with a bigger percentile of line rate, i.e., 99% of line rate the OSNT is enabled to capture few packets correctly having a greater percentile of error.

To understand the latency graphs, we are using a box-plot which shows the outlier,

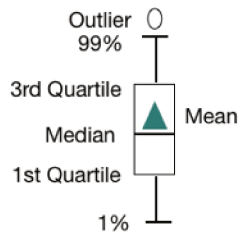


Figure 4.12: Boxplot for latency representation

average, mean median, 99% and 1% values as shown in Figure 4.12.
The impact of latency and throughput by using different burst sizes, FIB and packet sizes are presented in the next sub-sections. The testbed for latency measurements is shown in Figure 4.6a.

4.2.4.1 Impact of FIB and Packet Sizes

The box plot in Figures 4.13 and 4.14 shows the latency on BNG_UL use case with different FIB and packet sizes. MACSAD is configured with 32 of burst size, two cores and both DPDK and Netmap PktIO. From the results, as mention before, we can observe the linear increase of latency along with the packet size. According to Table 4.2, the percentage of latency increase for packet size 98B to 1518B is around 30% for all cases with different number of FIB entries.

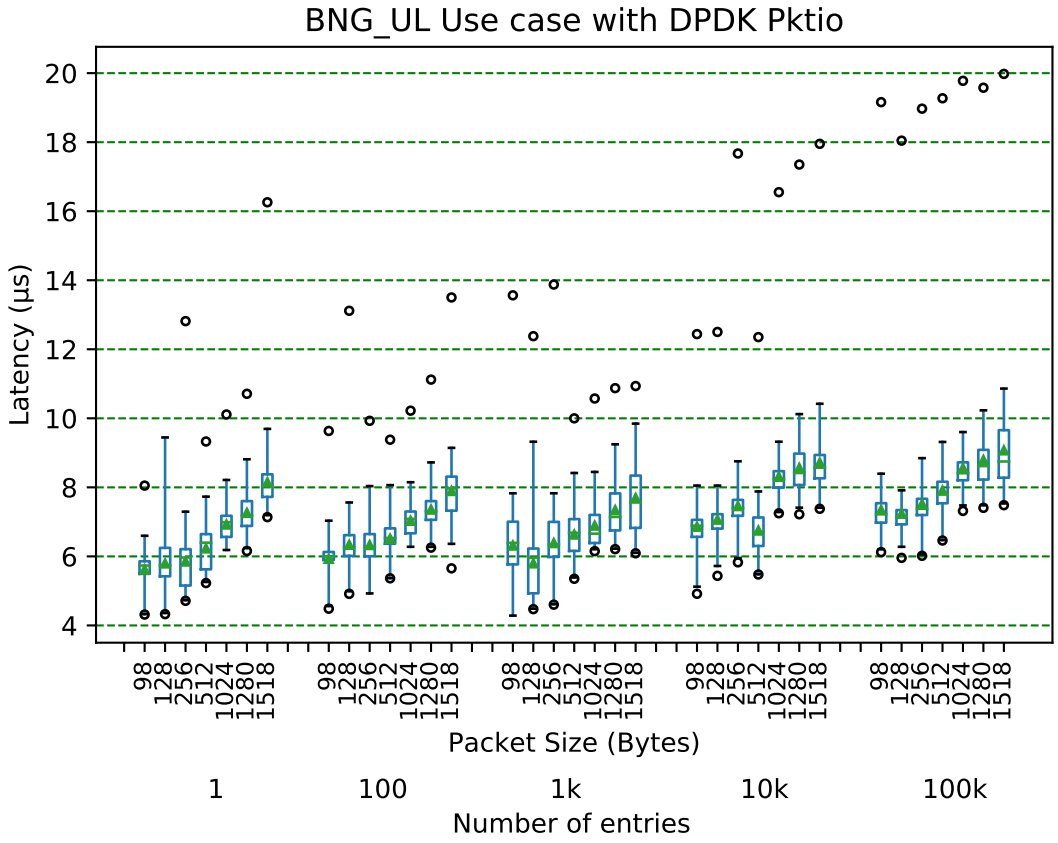


Figure 4.13: Impact of the number of FIB sizes in the Latency for BNG_UP with DPDK and burst size of 32.

Furthermore, with DPDK PktIO the increment in the percentage of latency from 1 to 100k flow entries is 30,7 % for 98B of packet size unlike the increase in the percentage of latency from 1 to 100k flow entries for 1518 packet size which increments 11.37 % according to Table 4.2. It makes sense because for small packet sizes with more FIB entries the CPU needs to process the packets faster, increasing also the latency in the Rx queue. Therefore the latency increase from 1-100K FIB entries is most for smaller packet sizes than bigger packet sizes.

On the other hand, according to table 4.3 The percentage of latency increase for packet size 98B to 1518B is around 16% for all cases with a different number of entries with Netmap PktIO. Also, the percentage of increment of latency from 1 to 100k flow entries is 14,9 % for smaller packet sizes unlike the increase in the percentage of latency from 1 to

Packet Size (Bytes)	1 entry (μs)	100 entries (μs)	1k entries (μs)	10k entries (μs)	100k entries (μs)	Latency increase from 1 to 100k entries (%)
98	5,62	5,93	6,32	6,86	7,34	30,70%
128	5,80	6,34	5,80	7,07	7,24	24,85%
256	5,85	6,33	6,39	7,46	7,52	28,56%
512	6,23	6,51	6,63	6,75	7,91	26,91%
1024	6,92	7,03	6,89	8,31	8,56	23,63%
1280	7,26	7,36	7,34	8,56	8,79	21,11%
1518	8,15	7,90	7,69	8,72	9,07	11,37%
Latency increase from 98B to 1518B (%).	44,96%	33,10%	21,81%	27,04%	23,53%	

Table 4.2: Latency results with a different FIB and packet sizes for BNG-UL use case with DPDK

100k flow entries for bigger packet sizes which increment at 11.61 %.

The maximum latency for BNG UL is with 1518 bytes of packet size and 100k FIB

Packet Size (Bytes)	1 entry (μs)	100 entries (μs)	1k entries (μs)	10k entries (μs)	100k entries (μs)	Latency increase from 1 to 100k entries (%)
98	15,41	16,49	16,70	16,71	17,70	14,91%
128	15,35	16,24	17,26	17,29	17,67	15,00%
256	15,40	16,78	17,45	17,62	17,82	15,74%
512	16,05	16,94	17,63	18,22	18,40	14,63%
1024	17,01	18,10	18,72	19,09	19,35	13,74%
1280	17,42	18,35	19,06	19,12	19,43	11,52%
1518	17,99	19,06	19,43	19,51	20,08	11,61%
Latency increase from 98B to 1518B (%).	16,80%	15,54%	16,32%	17,36%	13,45%	

Table 4.3: Latency results with different FIB sizes and packet sizes for the BNG_UL use case with Netmap

entries, being 20 μs for DPDK and 35.1 μs for Netmap.

All the maximum latency values always correspond to the first incoming packet. This is because on switch starting the CPU reads the routing table data on the RAM using the cache memory which is empty, thus it spends more time to access, find and modify the RAM data, therefore the time to data access residing in RAM is shortened to next packets by using the data in the cache memory. This cache influence analysis was explained in (GALLENMüLLER *et al.*, 2015).

4.2.4.2 Impact of burst Size

The ODP Libraries bring a burst processing mechanism to improve the packet processing at high rates. The burst size is essential as it determines how much packets waited within the Rx queue. This experiment varies the burst size from 8 to 256 and analyzes its influence on latency as described above in sub-section 4.2.4.

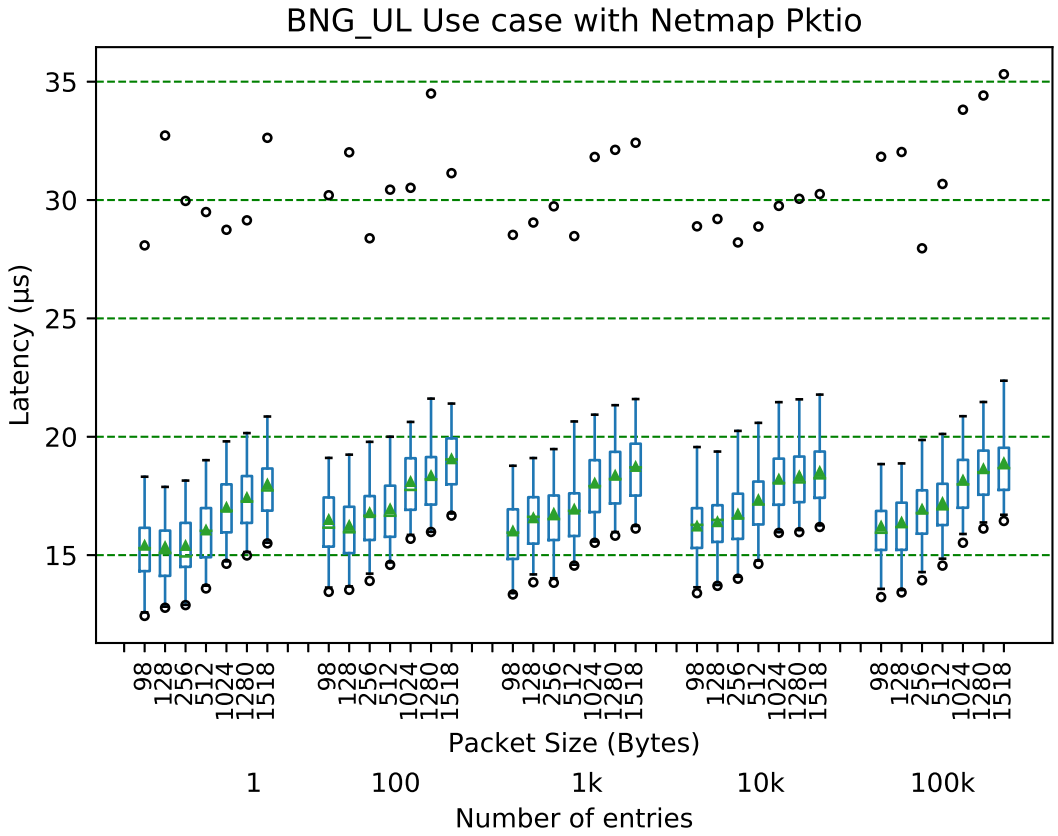


Figure 4.14: Impact of the number of FIB sizes in the Latency for BNG_UP with Netmap and burst size of 32.

Figures 4.15 and 4.16 show the BNG UP/DL latency for different burst sizes respectively. As expected the latency increases with bigger packet sizes for both DPDK and Netmap because bigger packet sizes originate overhead in forwarding processes by filling the Rx queues.

In Netmap the system is overloaded generating higher latency for all the burst sizes thus, due to the lower packet processing efficiency, the packets are accumulated in the Rx queue. Therefore, to generate a reduction of latency a big improvement could be to increase the burst size since the CPU can process more packets per second reducing packets in the queue and then the latency. The BNG_UP with Netmap achieves an average latency of 15 μs and 17 μs with a packet size of 98/1518 respectively and burst size of 256. Likewise, the average latency in BNG_DL achieves 15.2 μs and 17.7 μs with a packet size of 98/1518 respectively and burst size of 256. This value is lower than OVS implementation result in (ROTSOS *et al.*, 2012) which achieves an average latency of 35 μs.

Unlike Netmap, BNG S/W switch with DPDK achieves faster packet processing even with small burst sizes, then the latency increases slightly due to additional CPU cycles to handle the incoming packets from the Rx queue. It is confirmed in (MIAO *et al.*, 2016) where the average CPU cycles cost increases due to the increase of burst size for DPDK packet IO. At this point, the BNG_UP achieves an average latency of 7.02 μs and 8.7 μs with a

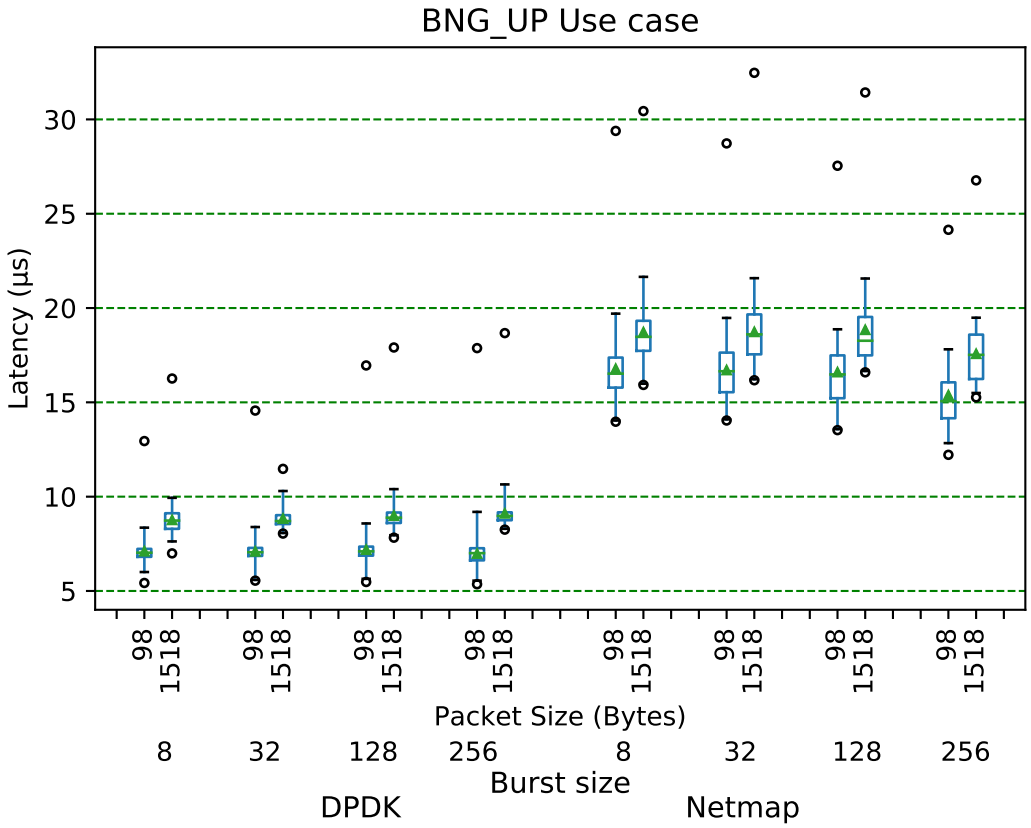


Figure 4.15: Impact of burst size in the Latency for BNG-UL with DPDK and Netmap.

packet size of 98/1518 respectively and a burst size of 8. Likewise, the average latency in BNG_DL achieves 6.6 μs and 8.8 μs with a packet size of 98/1518 respectively and burst size of 256. These latencies can be compared to other forwarding implementations and H/W switches: (ZAOSTROVNYKH *et al.*, 2017) measured a latency of 5.13 μs, 2-6 μs in (ROTSOS *et al.*, 2012) for hardware-based switch implementations. On the other hand, the maximum latency for BNG UP with DPDK is 17.8 μs and 18.6 μs for packet size of 98/1518 respectively with a burst size of 256, and 32.4 μs and 29.4 μs with Netmap for BNG DL for packet size of 98/1518 respectively with burst size of 8 and 32 respectively.

4.2.4.3 Impact of increasing the pipeline complexity, FIB and packet sizes

In this experiment, we have used different FIB sizes (1,100,1k,100k), smaller packet sizes (74/98B for BNG and 64B for L2, L3, and NAT) and bigger packet sizes (1518B) and configured MACSAD with 32 of burst size for L2, L3, NAT, and BNG use cases with DPDK PktIO. Figure 4.17 shows the impact of the increase of pipeline complexity based on uses cases above on which the BNG is composed too. We can observe that the latency is increased for complex use cases such as NAT and BNG. Also, the increment of latency is notable for smaller for complex use cases than simple use cases. i.e., The difference in latency between L2 and BNG-DL with one entry for 64/98 packet size is 1.41 μs which corresponds to an increment of 31% in the latency in comparison with the latency increment of L2-BNG with one entry for 1518 packet size which is 13%. Therefore,

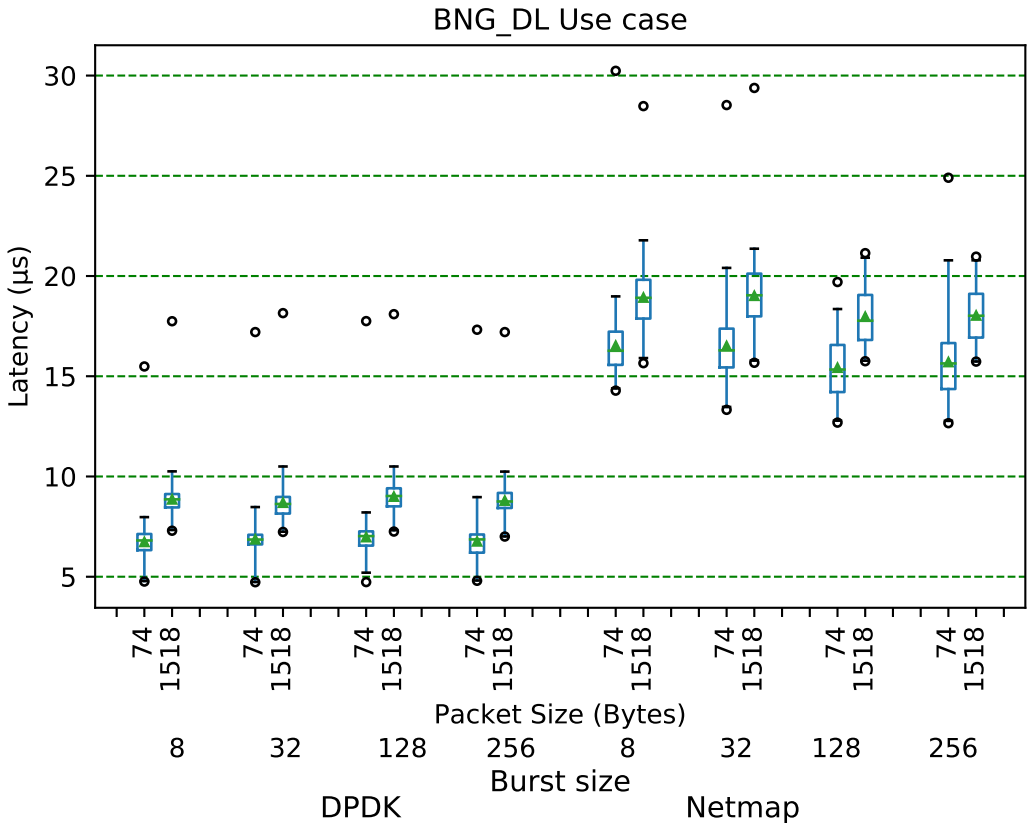


Figure 4.16: Impact of burst size in the Latency for BNG-DL with DPDK and Netmap.

the latency with smaller packet sizes in complex use cases is more affected than in simple use cases because complex use cases require more capacity for packet processing with smaller packet sizes.

4.3 Final Remarks

In this chapter, we presented the evaluation of the BNG S/W switch. In the functional evaluation a virtual environment was configured on which the BNG dataplane was targeted on the *P4 Simple Switch*. This software target let us create and run the BNG dataplane faster with vEths emulating a network scenario. Also, to generate the traffic traces and corresponding controller entries for all the experiments, Scapy Python libraries brought support for many protocols and packet sizes to validate the functionalities of the BNG dataplane. Later on, we targeted the BNG dataplane in an X86 Server by compiling it with MACSAD. Different experiments were conducted varying the workload presenting several results in throughput and latency metrics which were performed in two different test-beds with two traffic generators.

At the beginning of BNG latency measurements we found many issues (i.e., negative values) since OSNT was computing wrong latency values. In order to resolve it, we analyzed the main configuration parameters of OSNT such as IPG, time-stamp values, Tx/Rx interfaces and found that the OSNT script always writes the TX and RX time-stamp

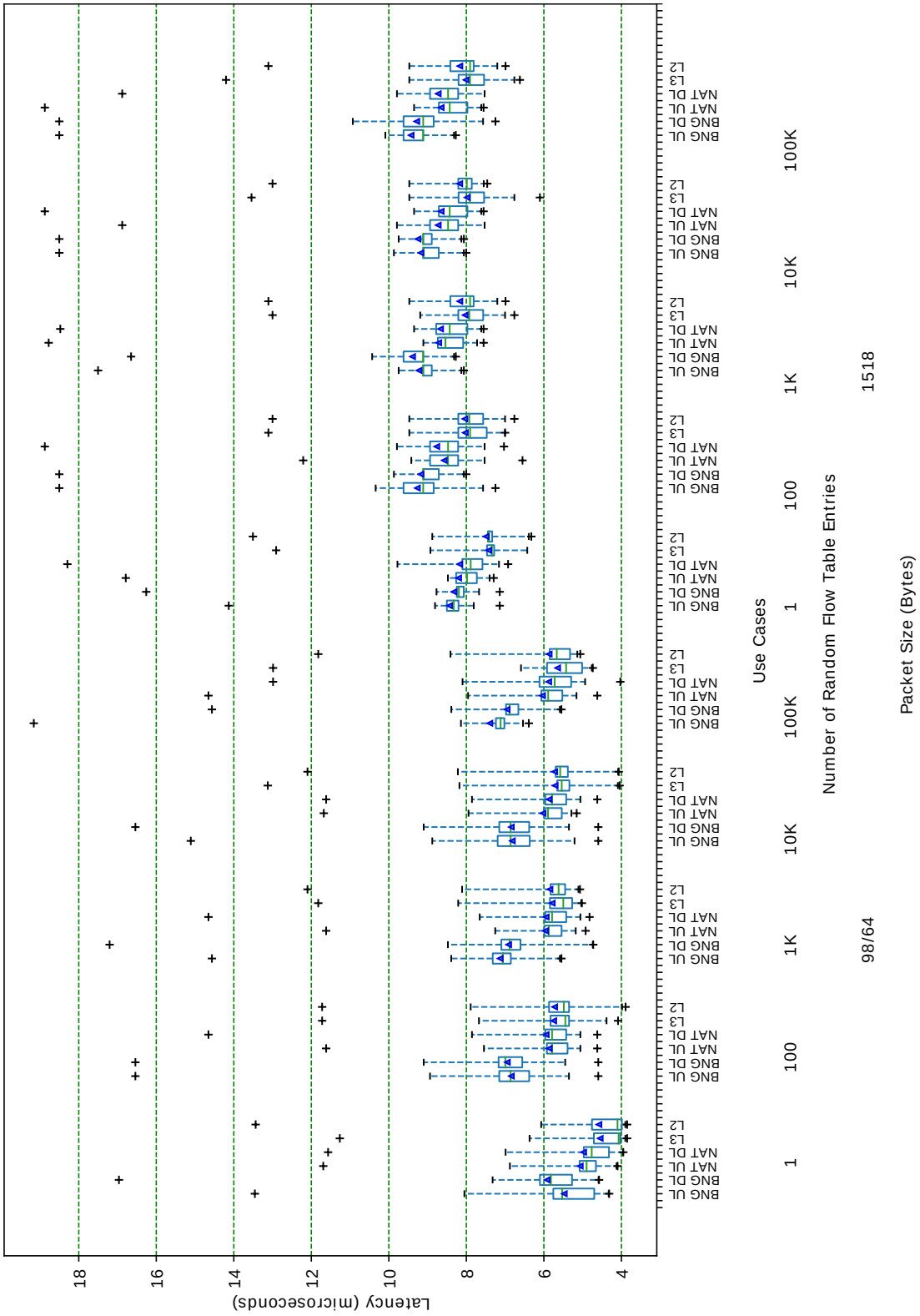


Figure 4.17: Latency for increasing pipeline complexity, FIB sizes and packet sizes with DPDK.

values on the packet in a static position. However, due to the BNG Encapsulation/Decap process, these values always are relocated. Therefore we created a method to calculate the timestamp without changing the OSNT script by adding a new timestamp header at

the beginning of the packet which is processed in the BNG along with the others headers. Then, as the new header never changes position, the timestamp values can be calculated in the same packet position resolving the issue.

We presented the results of performance at a high-speed network with an H/W vs. S/W based traffic generators. Although software-based solutions are convenient regarding deployment and costs they limit the measurements accuracy.

Configuring MACSAD with DPDK and larger burst sizes increases the performance slightly and also the average latency while MACSAD with NETMAP and larger burst sizes the impact in performance is notable by improving the packet processing and reducing the latency. These results lets us choose the best configuration of MACSAD according to an application requirement, i.e., larger batch sizes for applications with Netmap where higher throughput is critical. The best MACSAD configuration for the BNG performance is with two cores (2 for TX queues and 2 for RX queues) using 32 and 256 of burst size for DPDK and Netmap respectively.

We show that ODP with DPDK and Netmap PktIO help to improve the performance in P4 dataplanes with ODP alone being discarded because of low performance. In addition, we found that others configuration parameters such as CPU scaling, burst sizes, and network parameters must be taken into account when implementing these types of S/W switches. We have shown that without this information it is difficult to make better use of the H/W target and to optimize the system for each network application.

The performance results reported in Figure 4.9 using DPDK are similar to those in (ROBERTO *et al.*, 2013) and (DIETZ *et al.*, 2015) from the related work by using Click framework which can handle line rate for medium and bigger packet sizes (512-1518 Bytes) when using a single CPU core.

The performance results of simple and complex use cases from Figure 4.11 and 4.17 show that the performance is more affected in smaller packet sizes for complex uses cases than for simple uses cases. We can highlight that the average packet size in Internet Traffic according to (JURKIEWICZ; BORYŁO, 2018) is 870 Bytes in which our BNG performs well.

Our work gives an idea of which parameters can be varied to improve latency and throughput depending on the specific requirements of the data plane application.

Chapter 5

Conclusions and Future Work

In this work, we present the design, implementation, and evaluation of a BNG S/W switch using P4 a high-level programming language to specify the dataplane functionalities with flexibility along with protocol and target independence. Our implementation is the first implementation of BNG based in P4. The P4 codes of BNG, NAT, L2, and L3 are available in the public MACSAD use cases repository[1] for reproducibility purposes.

Our BNG prototype is validated on an x86 server achieving line rate at 10Gbps with medium and large packets (1024-1518), ratifying the support of MACSAD for complex dataplanes with programmability, portability, and high performance.

The MACSAD compiler offers support for both P4_16 and P4_14 versions by using an intermediate representation. Thus, the BNG functionalities are mapped on ODP APIs. We are currently working to put this support in the master branch of the MACSAD repository. The latest version of MACSAD with P4_16 support is on a separate MACSAD repository branch[2].

We present two evaluation scenarios; the first one brings a virtualized behavior which brings rapid development of P4 dataplanes by using *SimpleSwitch* target which is a complete switch to evaluate the BNG functionalities. The second evaluation scenario brings a realistic scenario by using MACSAD compiler to generate a BNG dataplane over an X86 Server.

The results obtained during this work help to clarify what the impact is of dataplane complexity in performance. The traffic generators are the tools to measure the performance concerning network metrics. Although the OSNT tool is an accurate system to measure latency and throughput it is more complex to install and run rather with NFPA which brings a moderate throughput accuracy, fast installation, and it is an excellent tool to reproduce the same testing scenario obtaining similar results.

We have exposed some specific platform parameters such as CPU, burst sizes, PktIO frameworks to improve the performance that must be taken into account to make optimal use of the target device. These platform parameters are key and are tailored according to the requirements of a network application either to optimize the throughput, latency, or both.

This work has contributed to two national conference publications, one IEEE Journal

[1] https://github.com/intrig-unicamp/macsad-usecases
[2] https://github.com/intrig-unicamp/macsad/tree/mac-16

publication, one mini-course in a national conference, and one demo in an international P4 Workshop (see Annex A).

As future work, we intend to implement and show the performance results of BNG in other targets such as VMs and ARM-based systems, and also add new features considering IPv6 deployment.

There are many opportunities for researchers to scale out and personalize the BNG instances, and to deploy different applications with different latency and throughput characteristics. We have identified some critical parameters related to the P4 dataplane as well as for the PktIO framework, burst sizes and CPU usages to optimize the performance and reliability. However, cache memory, RAM, queuing management parameters to name a few, can be explored to solve the network bottlenecks mainly on smaller packets size.

Soon the deployment of 40 and 100 Gbps networks will be common, on which lower latency will be required along with greater accuracy in terms of nanosecond accuracy at those speeds. Given the results, we envision that FPGA-based traffic generators will be a must for testing networks. Currently, we can find FPGAs such as Xilinx XC7Z030 which are capable of implementing both 10 and 40 Gb/s interfaces.

Bibliography

ALCATEL-LUCENT. Evolution of the Broadband Network Gateway. 2010. Cited on page 22.

ANTICHI, G.; SHAHBAZ, M.; GENG, Y.; ZILBERMAN, N.; COVINGTON, A.; BRUYERE, M.; MCKEOWN, N.; FEAMSTER, N.; FELDERMAN, B.; BLOTT, M.; MOORE, A. W.; OWEZARSKI, P. Osnt: open source network tester. *IEEE Network*, v. 28, n. 5, p. 6–12, Sep. 2014. ISSN 0890-8044. Cited 2 times on page(s) 10 and 30.

BALLANI, H.; COSTA, P.; KARAGIANNIS, T.; ROWSTRON, A. Towards predictable datacenter networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 41, n. 4, p. 242–253, ago. 2011. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2043164.2018465>. Cited on page 39.

BAREFOOT. *The Barefoot P4 Studio*. 2018. Accessed: 2018-10-15. Available from Internet: <https://www.barefootnetworks.com/products/brief-capilano/>. Cited on page 18.

BMV2. *The BMv2 Simple Switch target*. 2018. Accessed: 2018-08-18. Available from Internet: <https://github.com/p4lang/behavioral-model/blob/master/docs/simple_switch.md>. Cited on page 25.

BRADNER, S.; MCQUAID, J. *Benchmarking Methodology for Network Interconnect Devices*. United States: RFC Editor, 1999. Cited on page 48.

BUDIU, M.; DODD, C. The p416 programming language. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 51, n. 1, p. 5–14, set. 2017. ISSN 0163-5980. Available from Internet: <http://doi.acm.org/10.1145/3139645.3139648>. Cited 3 times on page(s) 10, 24, and 26.

CONSORTIUM, T. P. L. *P4_16 Language Specification*. 2017. Https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.html. Accessed: 2018-07-27. Available from Internet: <https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf>. Cited on page 25.

CSIKOR, L.; SZALAY, M.; SONKOLY, B.; TOKA, L. Nfpa: Network function performance analyzer. 2015. Cited 2 times on page(s) 10 and 31.

DIETZ, T.; BIFULCO, R.; MANCO, F.; MARTINS, J.; KOLBE, H. J.; HUICI, F. Enhancing the bras through virtualization. In: *Proceedings of the 2015 1st IEEE NetSoft*. [S.l.: s.n.], 2015. p. 1–5. Cited 2 times on page(s) 32 and 61.

ELTE. *HLIR for P4-16*. 2017. Https://github.com/P4ELTE/hlir16. Accessed: 2018-07-27. Available from Internet: <https://github.com/P4ELTE/hlir16>. Cited on page 40.

EMMERICH, P.; GALLENMüLLER, S.; ANTICHI, G.; MOORE, A. W.; CARLE, G. Mind the gap - a comparison of software packet generators. p. 191–203, May 2017. Cited on page 51.

FARINACCI, D.; LI, T.; HANKS, S.; MEYER, D.; TRAINA, P. *Generic Routing Encapsulation (GRE)*. [S.l.], 2000. Cited on page 38.

GALLENMüLLER, S.; EMMERICH, P.; WOHLFART, F.; RAUMER, D.; CARLE, G. Comparison of frameworks for high-performance packet io. In: *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. Washington, DC, USA: IEEE Computer Society, 2015. (ANCS '15), p. 29–38. ISBN 978-1-4673-6632-8. Available from Internet: <http://dl.acm.org/citation.cfm?id=2772722.2772729>. Cited on page 56.

HAN, B.; GOPALAKRISHNAN, V.; JI, L.; LEE, S. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, v. 53, n. 2, p. 90–97, Feb 2015. ISSN 0163-6804. Cited on page 19.

HEINANEN, J.; GUERIN, R. *A Two Rate Three Color Marker*. [S.l.], 1999. Cited on page 39.

HWANG, J.; RAMAKRISHNAN, K. K.; WOOD, T. Netvm: High performance and flexible networking using virtualization on commodity platforms. *IEEE Transactions on Network and Service Management*, v. 12, n. 1, p. 34–47, March 2015. ISSN 1932-4537. Cited on page 32.

INTEL. *Data Plane Development Kit*. 2018. Accessed: 2018-07-27. Available from Internet: <http://dpdk.org/>. Cited 2 times on page(s) 31 and 32.

JOSE, L.; YAN, L.; VARGHESE, G.; MCKEOWN, N. Compiling packet programs to reconfigurable switches. In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2015. (NSDI'15), p. 103–115. ISBN 978-1-931971-218. Available from Internet: <http://dl.acm.org/citation.cfm?id=2789770.2789778>. Cited on page 17.

JURKIEWICZ, G. R. P.; BORYłO, P. *Flow Length and Size Distributions in Campus Internet Traffic* . 2018. Accessed: 2018-12-18. Available from Internet: <https://arxiv.org/pdf/1809.03486.pdf>. Cited on page 61.

KAWASHIMA, R.; NAKAYAMA, H.; HAYASHI, T.; MATSUO, H. Evaluation of forwarding efficiency in nfv-nodes toward predictable service chain performance. *IEEE Transactions on Network and Service Management*, v. 14, n. 4, p. 920–933, Dec 2017. ISSN 1932-4537. Cited on page 54.

KREUTZ, D.; RAMOS, F. M. V.; VERíSSIMO, P. E.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, v. 103, n. 1, p. 14–76, Jan 2015. ISSN 0018-9219. Cited on page 17.

M. Shahbaz et al. PISCES: A Programmable, Protocol-Independent Software Switch. In: *ACM SIGCOMM*. [S.l.: s.n.], 2016. ISBN 978-1-4503-4193-6. Cited on page 32.

MACSAD. *MACSAD Github Repository*. 2018. Accessed: 2018-08-18. Available from Internet: <https://github.com/intrig-unicamp/macsad.git>. Cited on page 44.

MASUTANI, H.; NAKAJIMA, Y.; KINOSHITA, T.; HIBI, T.; TAKAHASHI, H.; OBANA, K.; SHIMANO, K.; FUKUI, M. Requirements and design of flexible nfv network infrastructure node leveraging sdn/openflow. p. 258–263, May 2014. Cited on page 32.

MEJIA, J. S. M.; FEFERMAN, D. L.; ROTHENBERG, C. E. Network Address Translation using a Programmable Dataplane Processor. *In 7º Workshop em Desempenho de Sistemas Computacionais e de Comunicação (Wperformance) at XXVIII Congresso da Sociedade Brasileira de Computação - CSBC 2018, Natal-RN, Brazil, 22 a 26 de Julho*, p. 4, 2018. Cited 3 times on page(s) 38, 52, and 53.

MEYER, T.; WOHLFART, F.; RAUMER, D.; WOLFINGER, B. E.; CARLE, G. Validated Model-Based Performance Prediction of Multi-Core Software Routers. *PIK - Praxis der Informationsverarbeitung und Kommunikation*, v. 37, n. 2, p. 93–107, 2014. ISSN 1865-8342. Cited on page 52.

MIAO, M.; CHENG, W.; REN, F.; XIE, J. Smart batching: A load-sensitive self-tuning packet i/o using dynamic batch sizing. In: *2016 IEEE 18th International Conference on High Performance Computing and Communications; IEEE 14th International Conference on Smart City; IEEE 2nd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. [S.l.: s.n.], 2016. p. 726–733. Cited on page 57.

MONKMAN, B. *The Emergence of the OpenDataPlaneTM Standard*. 2018. Accessed: 2018-08-30. Available from Internet: <https://community.arm.com/iot/b/blog/posts/the-emergence-of-the-opendataplane-standard>. Cited 2 times on page(s) 10 and 28.

NEMETH, B.; SIMONART, X.; OLIVER, N.; LAMOTTE, W. The limits of architectural abstraction in network function virtualization. p. 633–639, May 2015. ISSN 1573-0077. Cited on page 32.

NETWORKS, J. *Understanding Subscriber Management and BNG* . 2018. Accessed: 2018-12-18. Available from Internet: <https://www.juniper.net/documentation/en_US/design-and-architecture/service-provider-edge/information-products/topic-collections/understanding-subscriber-mgmt.pdf>. Cited on page 20.

ODP. *The ODP API Specification*. 2018. Accessed: 2018-08-18. Cited 3 times on page(s) 10, 28, and 31.

OECD. Improving Networks and Services Through Convergence discussion paper. *The Organization for Economic Co-operation and Development, Ministerial Meeting 2016*, 2016. Cited on page 17.

P, B.; D, D.; M, I.; MCKEOWN, N.; REXFORD, J.; SCHLESINGER, C.; TALAYCO, D.; VAHDAT, A.; VARGHESE, G.; W, D. Programming Protocol-Independent Packet Processors. 2013. ISSN 01464833. Available from Internet: <http://arxiv.org/abs/1312.1719>. Cited 4 times on page(s) 10, 17, 23, and 24.

P4C. *P4-16 compiler reference implementation*. 2018. Accessed: 2018-08-18. Available from Internet: <https://github.com/p4lang/p4c>. Cited 2 times on page(s) 17 and 26.

PATRA, P.; ROTHENBERG, C.; PONGRACZ, G. MACSAD: High performance dataplane applications on the move. *IEEE International Conference on High Performance Switching and Routing, HPSR*, v. 2017-June, 2017. ISSN 23255609. Cited 4 times on page(s) 12, 17, 27, and 53.

PATRA, P. G.; CESEN, F. R.; VALLEJO, J. S. M.; FEFERMAN, D. L.; CSIKOR, L.; ROTHENBERG, C. E.; PONGRÁCZ, G. Towards a Sweet Spot of Dataplane Programmability, Portability and Performance: On the Scalability of Multi-Architecture P4 Pipelines. *IEEE JSAC issue on Scalability Issues and Solutions for Software Defined Networks*, USA, sep 2018. Cited 3 times on page(s) 10, 27, and 53.

PFAFF, B.; PETTIT, J.; KOPONEN, T.; JACKSON, E. J.; ZHOU, A.; RAJAHALME, J.; GROSS, J.; WANG, A.; STRINGER, J.; SHELAR, P.; AMIDON, K.; CASADO, M. The design and implementation of open vswitch. In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2015. (NSDI'15), p. 117–130. ISBN 978-1-931971-218. Available from Internet: <http://dl.acm.org/citation.cfm?id=2789770.2789779>. Cited on page 32.

PLATFORMS, N. family of open-source networking. *NetFPGA-SUME Started Guide* . 2018. Accessed: 2018-08-18. Available from Internet: <https://github.com/NetFPGA/NetFPGA-public/wiki/Getting-Started-Guide>. Cited on page 30.

PLATFORMS, N. family of open-source networking. *OSNT for NetFPGA-SUME board* . 2018. Accessed: 2018-08-18. Available from Internet: <https://github.com/NetFPGA/OSNT-SUME-live>. Cited on page 30.

PONGRáCZ, G.; MOLNáR, L.; KIS, Z. L. Removing roadblocks from sdn: Openflow software switch performance on intel dpdk. p. 62–67, Oct 2013. ISSN 2379-0350. Cited on page 32.

POPA, L.; YALAGANDULA, P.; BANERJEE, S.; MOGUL, J.; TURNER, Y.; SANTOS, J. Elasticswitch: Practical work-conserving bandwidth guarantees for cloud computing. v. 43, p. 351–362, 08 2013. Cited on page 39.

PROJECT, N. *NetFPGA official website* . 2018. Accessed: 2018-08-18. Available from Internet: <https://netfpga.org>. Cited on page 30.

RAHIMI, R.; VEERARAGHAVAN, M.; NAKAJIMA, Y.; TAKAHASHI, H.; NAKAJIMA, Y.; OKAMOTO, S.; YAMANAKA, N. A high-performance openflow software switch. p. 93–99, June 2016. Cited on page 33.

RIZZO, L. Netmap: A novel framework for fast packet i/o. In: *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*. Berkeley, CA, USA: USENIX Association, 2012. (USENIX ATC'12), p. 9–9. Available from Internet: <http://dl.acm.org/citation.cfm?id=2342821.2342830>. Cited on page 31.

ROBERTO, B.; THOMAS, D.; H, F.; A, M.; M, J.; N, S.; K, H.-J. Rethinking Access Networks with High Performance Virtual Software BRASes. *EWSDN*, 2013. Cited 2 times on page(s) 32 and 61.

ROTSOS, C.; SARRAR, N.; UHLIG, S.; SHERWOOD, R.; MOORE, A. W. Oflops: An open framework for openflow switch evaluation. In: *Proceedings of the 13th International Conference on Passive and Active Measurement*. Berlin, Heidelberg: Springer-Verlag, 2012. (PAM'12), p. 85–95. ISBN 978-3-642-28536-3. Available from Internet: <http://dx.doi.org/10.1007/978-3-642-28537-0_9>. Cited 2 times on page(s) 57 and 58.

SHAHBAZ, M.; FEAMSTER, N. The case for an intermediate representation for programmable data planes. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. New York, NY, USA: ACM, 2015. (SOSR '15), p. 3:1–3:6. ISBN 978-1-4503-3451-8. Available from Internet: <http://doi.acm.org/10.1145/2774993.2775000>. Cited on page 33.

VOROS, P.; D, H.; KITLEI, R.; LESKO, D.; TEJFEL, M.; LAKI, S. T4P4S: A Target-independent Compiler for Protocol-independent Packet Processors. *IEEE International Conference on High Performance Switching and Routing, HPSR*, v. 2018-June, 2018. Cited 2 times on page(s) 17 and 33.

XILINX. *SDNet Development Environment*. Accessed: 2018-10-15. Available from Internet: <https://www.xilinx.com/products/design-tools/software-zone/sdnet.html>. Cited on page 17.

ZAOSTROVNYKH, A.; PIRELLI, S.; PEDROSA, L.; ARGYRAKI, K.; CANDEA, G. A formally verified nat. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 141–154. ISBN 978-1-4503-4653-5. Available from Internet: <http://doi.acm.org/10.1145/3098822.3098833>. Cited on page 58.

Appendix A

Publications

Main author of two papers in Brazilian conferences, co-author of a paper published in the IEEE COMSOC jornal, co-author of a demo paper presented in the P4 Workshop and co-author of a Mini-course published during this work and listed accordingly below.

1. Juan Sebastian Mejia Vallejo , Christian Esteve Rothenberg. "Broadband Network Gateway implementation using a programmable data plane processor". In X DCA/FEEC/University of Campinas (UNICAMP) Workshop (EADCA), Campinas, Brazil, October 26-27, 2017

2. Juan Sebastian Mejia Vallejo, Christian Esteve Rothenberg. "Network Address Translation using a Programmable Dataplane Processor". In 7º Workshop em Desempenho de Sistemas Computacionais e de Comunicação (Wperformance) XXVIII Congresso da Sociedade Brasileira de Computação - CSBC 2018, Natal, 22 to 26 of Jule, 2018.

3. P. G. Patra, F. R. Cesen, J. M. Vallejo, D. L. Feferman, L. Csikor, C. E. Rothenberg, and G. Pongrácz, “Towards a Sweet Spot of Dataplane Programmability, Portability and Performance: On the Scalability of Multi-Architecture P4 Pipelines,” IEEE JSAC issue on Scalability Issues and Solutions for Software Defined Networks, 2018.

4. D. L. Feferman, J. M. Vallejo, N. F. Saraiva, and C. E. Rothenberg, “Uma Nova Revolucao em Redes: Programacao do Plano de Dados com P4,” in Escola Regional de Informátiva do Piauí (ERIPI), Teresina, Brazil, 2018.

5. P. G. Patra, F. R. Cesen, J. M. Vallejo, D. L. Feferman, C. E. Rothenberg, and G. Pongrácz, “MACSAD: An Exemplar Realization of Multi-Architecture P4 Pipelines”. In 5th P4 Workshop, in Stanford University, Palo Alto CA, USA on June 5th and 6th, 2018.

Appendix B

BNG P4 graphs

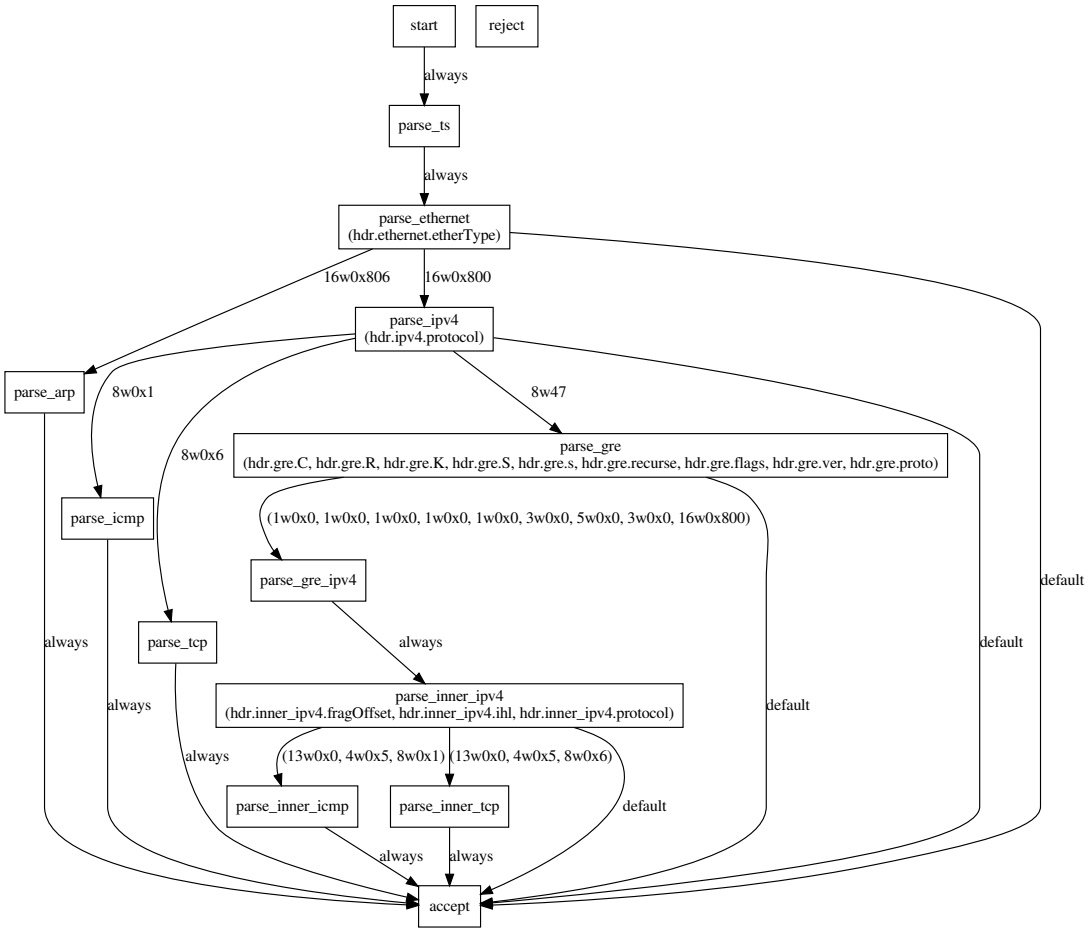


Figure B.1: Parser graph from P4 BNG program with time-stamp header.

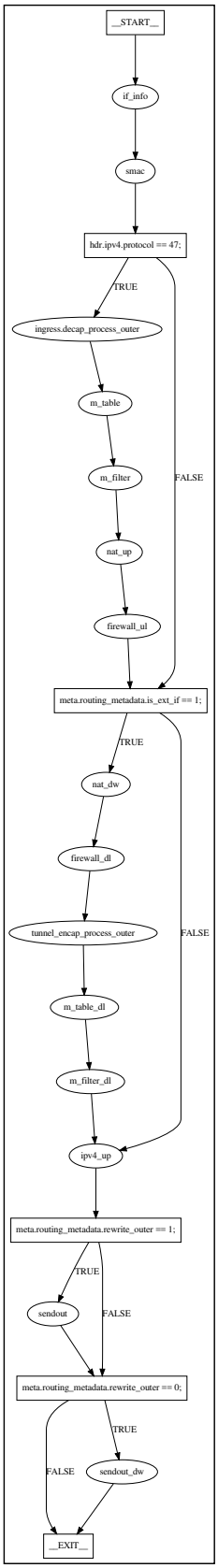


Figure B.2: Ingress graph from P4 BNG program.

Appendix C

ODP functions

```
void setValid(packet_descriptor_t* p, header_instance_t hdr_prefix)
{
    debug("calling setValid function\n");
    if(p->headers[hdr_prefix].pointer == NULL) {
        uint16_t len = header_instance_byte_width[hdr_prefix];
        uint32_t new = odp_packet_headroom (*((odp_packet_t *)(p->wrapper)));
        char* address = odp_packet_push_head(*((odp_packet_t *)(p->wrapper)), len);  //

        p->headers[hdr_prefix] =
            (header_descriptor_t) {
                .type = hdr_prefix,
                .pointer = address,
                .length = len
            };
    } else {
        debug("Cannot add a header instance already present in the packet\n");
    }
}
```

```
void setInvalid(packet_descriptor_t* p, header_instance_t hdr_prefix)
{
    debug("calling setInvalid function \n");
    if(p->headers[hdr_prefix].pointer == NULL) {
        printf("Cannot remove a header instance not present in the packet\n");
    }
    else {
        uint16_t len = header_instance_byte_width[hdr_prefix];
        char* address =  odp_packet_pull_head(*((odp_packet_t *)(p->wrapper)), len);       //

      p->headers[hdr_prefix] =
          (header_descriptor_t) {
                .type = hdr_prefix,
                .pointer = address,
                .length = len
            };

}
}
```

```
void mark_to_drop(packet_descriptor_t* p, header_instance_t hdr_prefix)
{
    debug("calling DROP function... \n");
    p->dropped = 1;

}
```

Appendix D

P4 code of BNG

```
#include <core.p4>
#include <v1model.p4>
#include "include/standard_headers.p4"

/************************  C O N S T A N T S  *****************************/
const bit<16> ETHERTYPE_IPV4 = 0x0800;
const bit<16> ETHERTYPE_ARP  = 0x0806;
const bit<8>  IPPROTO_ICMP   = 0x01;

/************************  H E A D E R S  *********************************/

const bit<16> ARP_HTYPE_ETHERNET = 0x0001;
const bit<16> ARP_PTYPE_IPV4     = 0x0800;
const bit<8>  ARP_HLEN_ETHERNET  = 6;
const bit<8>  ARP_PLEN_IPV4      = 4;


struct headers {
    ethernet_t   ethernet;
    ethernet_t   outer_ethernet;
    ethernet_t   ethernet_decap;
    arp_t        arp;
    ipv4_t       ipv4;
    ipv4_t       outer_ipv4;

    gre_t        gre;
    tcp_t        tcp;
    icmp_t       icmp;
    @name("inner_ipv4")
    ipv4_t       inner_ipv4;
    @name("inner_ethernet")
    ethernet_t   inner_ethernet;
    @name("inner_tcp")
    tcp_t        inner_tcp;
    @name("inner_icmp")
    icmp_t       inner_icmp;
}

 struct meta_ipv4_t {
     bit<4>       version;
     bit<4>       ihl;
     bit<8>       diffserv;
     bit<16>      totalLen;
     bit<16>      identification;
     bit<3>       flags;
     bit<13>      fragOffset;
     bit<8>       ttl;
     bit<8>       protocol;
     bit<16>      hdrChecksum;
     bit<32>      srcAddr;
     bit<32>      dstAddr;
 }
/************************  M E T A D A T A  *******************************/
struct routing_metadata_t {
    bit<32> nhgroup;

    bit<32> dst_ipv4;
    bit<32> src_ipv4;
    bit<48>  mac_da;
    bit<48>  mac_sa;
    bit<9>   egress_port;
    bit<48>  my_mac;

    bit<32> nhop_ipv4;
    bit<1>  do_forward;
    bit<1>  rewrite_outer;
```

```
67      bit<16> tcp_sp;
68      bit<16> tcp_dp;
69
70      bit<8>  if_index;
71      bit<32> if_ipv4_addr;
72      bit<48> if_mac_addr;
73      bit<1>  is_ext_if;
74
75      bit<32> tunnel_id;
76      bit<5>  ingress_tunnel_type;
77      bit<1>  tcp_inner_en;
78      bit<16> lkp_inner_l4_sport;
79      bit<16> lkp_inner_l4_dport;
80
81      bit<32>  dst_inner_ipv4;
82      bit<32>  src_inner_ipv4;
83
84      bit<32> meter_tag;
85
86  }
87
88  struct metadata {
89      @name(".routing_metadata")
90      routing_metadata_t routing_metadata;
91      @name(".meta_ipv4")
92      meta_ipv4_t meta_ipv4 ;
93
94  }
95
96
97
98  /***********************  P A R S E R  **********************************/
99  parser ParserImpl(packet_in packet, out headers hdr, inout metadata meta, inout standard_metadata_t standard_metadata)
        {
100     @name(".start") state start {
101         transition parse_ethernet;
102     }
103     @name ("parse_ethernet") state parse_ethernet {
104         packet.extract(hdr.ethernet);
105         transition select(hdr.ethernet.etherType) {
106             ETHERTYPE_IPV4 : parse_ipv4;
107             ETHERTYPE_ARP  : parse_arp;
108             default        : accept;
109         }
110     }
111     @name ("parse_arp") state parse_arp {
112         packet.extract(hdr.arp);
113         transition accept;
114     }
115
116     @name("parse_ipv4") state parse_ipv4 {
117         packet.extract(hdr.ipv4);
118         transition select(hdr.ipv4.protocol) {
119             IPPROTO_ICMP : parse_icmp;
120             8w0x6 :         parse_tcp;
121             8w47 :         parse_gre;
122             default      : accept;
123         }
124     }
125     @name("parse_icmp") state parse_icmp {
126         packet.extract(hdr.icmp);
127         transition accept;
128     }
129     @name("parse_tcp") state parse_tcp {
130         packet.extract<tcp_t>(hdr.tcp);
131         transition accept;
132     }
133     @name("parse_gre") state parse_gre {
134         packet.extract<gre_t>(hdr.gre);
135         transition select(hdr.gre.C, hdr.gre.R, hdr.gre.K, hdr.gre.S, hdr.gre.s, hdr.gre.recurse, hdr.gre.flags, hdr.
              gre.ver, hdr.gre.proto) {
136             (1w0x0, 1w0x0, 1w0x0, 1w0x0, 3w0x0, 5w0x0, 3w0x0, 16w0x800): parse_gre_ipv4;
137             default: accept;
138         }
139     }
140     @name(".parse_gre_ipv4") state parse_gre_ipv4 {
141         transition parse_inner_ipv4;
142     }
143
144     @name(".parse_inner_ipv4") state parse_inner_ipv4 {
145         packet.extract(hdr.inner_ipv4);
146         transition select(hdr.inner_ipv4.fragOffset, hdr.inner_ipv4.ihl, hdr.inner_ipv4.protocol) {
147             (13w0x0, 4w0x5, 8w0x1): parse_inner_icmp;
148             (13w0x0, 4w0x5, 8w0x6): parse_inner_tcp;
149             default: accept;
150         }
151     }
152     @name(".parse_inner_icmp") state parse_inner_icmp {
153         packet.extract(hdr.inner_icmp);
154         transition accept;
```

```
155     }
156
157     @name(".parse_inner_tcp") state parse_inner_tcp {
158         packet.extract(hdr.inner_tcp);
159         transition accept;
160     }
161
162     @name(".parse_inner_ethernet") state parse_inner_ethernet {
163         packet.extract(hdr.inner_ethernet);
164         transition select(hdr.inner_ethernet.etherType) {
165             16w0x800: parse_inner_ipv4;
166             default: accept;
167         }
168     }
169 }
170 @name("mac_learn_digest") struct mac_learn_digest {
171     bit<8> in_port;
172     bit<48> mac_sa;
173 }
174
175
176 /**************  I N G R E S S   P R O C E S S I N G   *****************/
177 control ingress(inout headers hdr, inout metadata meta, inout standard_metadata_t standard_metadata) {
178
179     /******************* Drop  *************************/
180      @name(".nop")action nop() {
181         meta.routing_metadata.mac_da = hdr.ethernet.dstAddr;
182     }
183      @name(".drop")action drop() {
184         mark_to_drop();
185
186     }
187     /***************************** set IF info and others  *************************/
188
189     @name(".set_if_info") action set_if_info(bit<1> is_ext) {
190
191
192         meta.routing_metadata.mac_da = hdr.ethernet.dstAddr;
193         meta.routing_metadata.mac_sa = hdr.ethernet.srcAddr;
194         meta.routing_metadata.if_ipv4_addr = 0x7fef4800 ;
195         meta.routing_metadata.if_mac_addr = 0x010101010100;
196         meta.routing_metadata.is_ext_if = is_ext;
197
198     }
199     @name(".if_info") table if_info {
200         key = { meta.routing_metadata.if_index: exact;}
201         actions = {   drop;  set_if_info; }
202     default_action = drop();
203
204     }
205     /***************************** process mac learn  *****************************/
206
207     @name(".generate_learn_notify") action generate_learn_notify() {
208         digest<mac_learn_digest>(32w1024, {meta.routing_metadata.if_index, hdr.ethernet.srcAddr });
209     }
210     @name(".smac") table smac {
211         actions = {
212             generate_learn_notify;
213         }
214         key = {
215             hdr.ethernet.srcAddr: exact;
216         }
217         size = 512;
218     }
219
220     /***************************** tunnel control decap ****************************/
221     @name(".decap_gre_inner_ipv4") action decap_gre_inner_ipv4(bit <32> tunnel_id) {
222
223           /* new decap */
224
225           /*inner metadata */
226           meta.meta_ipv4.version        = hdr.inner_ipv4.version        ;
227           meta.meta_ipv4.ihl            = hdr.inner_ipv4.ihl            ;
228           meta.meta_ipv4.diffserv       = hdr.inner_ipv4.diffserv       ;
229           meta.meta_ipv4.totalLen       = hdr.inner_ipv4.totalLen       ;
230           meta.meta_ipv4.identification = hdr.inner_ipv4.identification ;
231           meta.meta_ipv4.flags          = hdr.inner_ipv4.flags          ;
232           meta.meta_ipv4.fragOffset     = hdr.inner_ipv4.fragOffset     ;
233           meta.meta_ipv4.ttl            = hdr.inner_ipv4.ttl            ;
234           meta.meta_ipv4.protocol       = hdr.inner_ipv4.protocol       ;
235           meta.meta_ipv4.hdrChecksum    = hdr.inner_ipv4.hdrChecksum    ;
236           meta.meta_ipv4.srcAddr        = hdr.inner_ipv4.srcAddr        ;
237           meta.meta_ipv4.dstAddr        = hdr.inner_ipv4.dstAddr        ;
238
239           meta.routing_metadata.tunnel_id = tunnel_id;
240           meta.routing_metadata.dst_ipv4 = hdr.inner_ipv4.dstAddr;
241
242
243      standard_metadata.egress_port = 1;
244
```

```
245             meta.routing_metadata.is_ext_if = 0;
246
247     }
248
249     @name("decap_process_outer") table decap_process_outer {
250         actions = {decap_gre_inner_ipv4; drop; }
251         key ={ hdr.ethernet.srcAddr: exact;}
252
253         size = 1024;
254         default_action = drop();
255     }
256     /***************************** firewall UL control *****************************/
257     @name(".firewall_ul") table firewall_ul {
258         actions = { drop; nop; }
259         key = {
260             hdr.ipv4.dstAddr   : exact;
261             /*hdr.tcp.dstPort    : exact;*/
262         }
263         size = 128;
264         default_action = nop();
265     }
266
267     /***************************** Nat control ***************************************/
268     @name(".nat_hit_int_to_ext") action nat_hit_int_to_ext(bit<32> srcAddr, bit<16> srcPort) {
269         meta.routing_metadata.rewrite_outer = 1w1;
270         hdr.inner_ipv4.srcAddr= srcAddr;
271         hdr.inner_tcp.srcPort = srcPort;
272     }
273
274     @name(".nat_up") table nat_up {
275         actions = { drop; nat_hit_int_to_ext; }
276         key = { hdr.inner_ipv4.srcAddr: exact;}
277         size = 1024;
278         default_action = drop();
279     }
280
281     @name(".nat_hit_ext_to_int") action nat_hit_ext_to_int(bit<32> dstAddr) {
282         meta.routing_metadata.rewrite_outer = 1w0;
283         meta.routing_metadata.dst_ipv4 = dstAddr; /* to lpm */
284         hdr.ipv4.dstAddr = dstAddr;
285     }
286     @name(".nat_dw") table nat_dw {
287         actions = { drop; nat_hit_ext_to_int;  }
288         key = { hdr.tcp.dstPort: exact;}
289         size = 1024;
290         default_action = drop();
291     }
292
293     /***************************** tunnel control encap *****************************/
294
295     @name(".ipv4_gre_rewrite") action ipv4_gre_rewrite(bit<32> gre_dstAddr) {
296       hdr.ethernet.setInvalid();
297       hdr.gre.setValid();
298       hdr.gre.proto = 16w0x800;
299
300       meta.meta_ipv4.version        = hdr.ipv4.version        ;
301       meta.meta_ipv4.ihl            = hdr.ipv4.ihl            ;
302       meta.meta_ipv4.diffserv       = hdr.ipv4.diffserv       ;
303       meta.meta_ipv4.totalLen       = hdr.ipv4.totalLen       ;
304       meta.meta_ipv4.identification = hdr.ipv4.identification ;
305       meta.meta_ipv4.flags          = hdr.ipv4.flags          ;
306       meta.meta_ipv4.fragOffset     = hdr.ipv4.fragOffset     ;
307       meta.meta_ipv4.ttl            = hdr.ipv4.ttl            ;
308       meta.meta_ipv4.protocol       = hdr.ipv4.protocol       ;
309       meta.meta_ipv4.hdrChecksum    = hdr.ipv4.hdrChecksum    ;
310       meta.meta_ipv4.srcAddr        = hdr.ipv4.srcAddr        ;
311       meta.meta_ipv4.dstAddr        = hdr.ipv4.dstAddr        ;
312
313       hdr.outer_ipv4.setValid();
314       hdr.outer_ipv4.srcAddr  = 0x04000001;
315       hdr.outer_ipv4.dstAddr  = gre_dstAddr;
316       hdr.outer_ipv4.protocol = 47;
317       hdr.outer_ipv4.version        =  meta.meta_ipv4.version        ;
318       hdr.outer_ipv4.ihl            =  meta.meta_ipv4.ihl            ;
319       hdr.outer_ipv4.diffserv       =  meta.meta_ipv4.diffserv       ;
320       hdr.outer_ipv4.totalLen       =  meta.meta_ipv4.totalLen       ;
321       hdr.outer_ipv4.identification =  meta.meta_ipv4.identification ;
322       hdr.outer_ipv4.flags          =  meta.meta_ipv4.flags          ;
323       hdr.outer_ipv4.fragOffset     =  meta.meta_ipv4.fragOffset     ;
324       hdr.outer_ipv4.ttl            =  meta.meta_ipv4.ttl            ;
325
326       hdr.outer_ethernet.setValid();
327       hdr.outer_ethernet.dstAddr = 0x000000000001;
328       hdr.outer_ethernet.srcAddr = 0x000000000002;
329       hdr.outer_ethernet.etherType = 16w0x800;
330       standard_metadata.egress_port = 1;
331       meta.routing_metadata.dst_ipv4= hdr.outer_ipv4.dstAddr;
332       meta.routing_metadata.rewrite_outer = 0;
333     }
334
```

```
335       @name(".tunnel_encap_process_outer") table tunnel_encap_process_outer {
336         actions = {ipv4_gre_rewrite; drop; }
337         key = {  hdr.ipv4.dstAddr       : exact; }   /*  here must be ID */
338         size = 128;
339     }
340     /***************************** firewall DW control *****************************/
341     @name(".firewall_dl") table firewall_dl {
342         actions = {
343              drop; nop;
344         }
345         key = {
346             hdr.ipv4.dstAddr   : exact;
347         }
348         size = 128;
349         default_action = nop();
350     }
351
352     /************** forwarding ipv4 ******************/
353
354
355     @name(".set_nhop") action set_nhop(bit<32> nhop_ipv4, bit<9> port) {
356         /*standard_metadata.egress_port =  port;  */
357         hdr.inner_ipv4.ttl = hdr.ipv4.ttl + 8w255;
358     }
359
360     @name(".ipv4_up") table ipv4_up {
361         key = {meta.routing_metadata.dst_ipv4 : lpm;}
362         actions = { set_nhop; drop;  }
363     }
364     @name(".rewrite_src_mac") action rewrite_src_mac(bit<48> src_mac) {
365
366         hdr.ethernet.setInvalid();
367            hdr.ethernet_decap.setValid();
368            hdr.ethernet_decap.dstAddr =  meta.routing_metadata.mac_da;
369            hdr.ethernet_decap.srcAddr =  src_mac;
370            hdr.ethernet_decap.etherType = 16w0x800;
371     }
372
373     @name(".sendout") table sendout {
374         actions = {drop; rewrite_src_mac; }
375         key = {  standard_metadata.egress_port: exact; }
376         size = 512;
377     }
378
379     @name(".rewrite_src_mac_dw") action rewrite_src_mac_dw(bit<48> src_mac) {
380            hdr.outer_ethernet.dstAddr =  meta.routing_metadata.mac_da;
381            hdr.outer_ethernet.srcAddr =  src_mac;
382            hdr.outer_ethernet.etherType = 16w0x800;
383     }
384
385     @name(".sendout_dw") table sendout_dw {
386         actions = {drop; rewrite_src_mac_dw; }
387         key = {  standard_metadata.egress_port: exact; }
388         size = 512;
389     }
390
391
392     /***************************** process meter dl  *****************************/
393
394     @name(".my_meter") meter(32w16384, MeterType.packets) my_meter_dl;
395
396     @name(".m_action_dl") action m_action_dl(bit<32> meter_idx) {
397         my_meter_dl.execute_meter((bit<32>)meter_idx, meta.routing_metadata.meter_tag);
398         standard_metadata.egress_spec = 9w2;
399     }
400     @name(".m_filter_dl") table m_filter_dl {
401         actions = {drop; nop; }
402         key = { meta.routing_metadata.meter_tag: exact;}
403         size = 16;
404     }
405     @name(".m_table_dl") table m_table_dl {
406         actions = {m_action_dl; nop; }
407         key = { hdr.ethernet.srcAddr: exact;}
408         size = 16384;
409     }
410
411     /***************************** process meter UL  *****************************/
412     @name(".my_meter_ul") meter(32w16384, MeterType.packets) my_meter;
413     @name(".m_action") action m_action(bit<32> meter_idx) {
414         my_meter.execute_meter((bit<32>)meter_idx, meta.routing_metadata.meter_tag);
415         standard_metadata.egress_spec = 9w2;
416     }
417     @name(".m_filter") table m_filter {
418         actions = {drop; nop; }
419         key = { meta.routing_metadata.meter_tag: exact;}
420         size = 16;
421     }
422     @name(".m_table") table m_table {
423         actions = {m_action; nop; }
424         key = { hdr.ethernet.srcAddr: exact;}
```

```
425         size = 16384;
426     }
427
428
429
430
431
432
433     /************** APPLY ******************/
434     apply {
435         if_info.apply();
436         smac.apply();
437         /* ------ decap-------  */
438         if(hdr.ipv4.protocol== 8w47){
439            decap_process_outer.apply();
440            m_table.apply();
441            m_filter.apply();
442            nat_up.apply();
443            firewall_ul.apply();
444         }
445
446
447         if(meta.routing_metadata.is_ext_if == 1){
448            nat_dw.apply();
449            firewall_dl.apply();
450            tunnel_encap_process_outer.apply();
451            m_table_dl.apply();
452            m_filter_dl.apply();
453
454         }
455
456            ipv4_up.apply();
457
458
459         if (meta.routing_metadata.rewrite_outer == 1w1) {sendout.apply(); }
460         if (meta.routing_metadata.rewrite_outer == 1w0) {sendout_dw.apply(); }
461
462     }
463
464
465 }
466
467 control egress(inout headers hdr, inout metadata meta, inout standard_metadata_t standard_metadata) {
468     apply {        }
469 }
470
471 control DeparserImpl(packet_out packet, in headers hdr) {
472     apply {    }
473 }
474 /********   CHECKSUM VERIFICATION ************/
475 control verifyChecksum(inout headers hdr, inout metadata meta) {
476     apply {    }
477 }
478
479 control computeChecksum(inout headers hdr, inout metadata meta) {
480     apply {    }
481 }
482
483 V1Switch(ParserImpl(), verifyChecksum(), ingress(), egress(), computeChecksum(), DeparserImpl()) main;
```

Appendix E

BNG Controller

E.1 BNG Controller C-based code for MACSAD

The BNG controller is divided in UL and DL for purpose of the performance evaluation.

BNG controller for UL

```
#include "controller.h"
#include "messages.h"
#include <unistd.h>
#include <stdio.h>
#include <string.h>

#define MAX_MACS 1000000

controller c;

uint8_t macs[MAX_MACS][6];
uint8_t portmap[MAX_MACS];
uint8_t ips[MAX_MACS][4];
uint8_t ipd[MAX_MACS][4];
int mac_count = -1;
uint8_t stcp_txt[MAX_MACS][2];
uint8_t ips_new[MAX_MACS][4];

int read_macs_and_ports_from_file(char *filename) {
  FILE *f;
  char line[200];
  int values[6];
  int values_ip[4];
  int values_ip2[4];
  int i;
  int stcp;
  int port;

  f = fopen(filename, "r");
  if (f == NULL) return -1;

  while (fgets(line, sizeof(line), f)) {
    line[strlen(line)-1] = '\0';
    if (16 == sscanf(line, "%x:%x:%x:%x:%x:%x %d.%d.%d.%d %d.%d.%d.%d %d %d %d %d",
          &values[0], &values[1], &values[2],
          &values[3], &values[4], &values[5],
          &values_ip[0], &values_ip[1], &values_ip[2], &values_ip[3],
          &values_ip2[0], &values_ip2[1], &values_ip2[2], &values_ip2[3],
                                    &stcp, &port
                ) )
    {
      if (mac_count==MAX_MACS-1)
      {
        printf("Too many entries...\n");
        break;
      }

      ++mac_count;
      for( i = 0; i < 6; ++i )
        macs[mac_count][i] = (uint8_t) values[i];
      for( i = 0; i < 4; ++i )
        ips[mac_count][i] = (uint8_t) values_ip[i];
      for( i = 0; i < 4; ++i )
        ipd[mac_count][i] = (uint8_t) values_ip2[i];

      stcp_txt[mac_count][1] = (uint8_t) stcp;
      stcp_txt[mac_count][0] = 0;
```

```
58
59                          ips_new[mac_count][0] = (uint8_t) values_ip2[3];
60                          ips_new[mac_count][1] = (uint8_t) values_ip2[2];
61                          ips_new[mac_count][2] = (uint8_t) values_ip2[1];
62                          ips_new[mac_count][3] = (uint8_t) values_ip2[0];
63
64                  }else {
65        printf("Wrong format error in line %d : %s\n", mac_count+2, line);
66        fclose(f);
67        return -1;
68      }
69
70    }
71
72    fclose(f);
73    return 0;
74  }
75
76
77  void set_default_action_ipv4_forward()
78  {
79    char buffer[2048];
80    struct p4_header* h;
81    struct p4_set_default_action* sda;
82    struct p4_action* a;
83
84    h = create_p4_header(buffer, 0, sizeof(buffer));
85    sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
86          strcpy(sda->table_name, "ipv4_forward");
87
88    a = &(sda->action);
89    strcpy(a->description.name, "drop");
90
91    netconv_p4_header(h);
92    netconv_p4_set_default_action(sda);
93    netconv_p4_action(a);
94    send_p4_msg(c, buffer, sizeof(buffer));
95
96    printf("########## \n");
97    printf("\n");
98    printf("Table name: %s\n", sda->table_name);
99    printf("### Default action: %s\n", a->description.name);
100 }
101 void set_default_action_nat_up()
102 {
103   char buffer[2048];
104   struct p4_header* h;
105   struct p4_set_default_action* sda;
106   struct p4_action* a;
107
108   h = create_p4_header(buffer, 0, sizeof(buffer));
109
110   sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
111         strcpy(sda->table_name, "nat_up");
112
113   a = &(sda->action);
114   strcpy(a->description.name, "drop");
115
116   netconv_p4_header(h);
117   netconv_p4_set_default_action(sda);
118   netconv_p4_action(a);
119
120   send_p4_msg(c, buffer, sizeof(buffer));
121
122   printf("\n");
123   printf("Table name: %s\n", sda->table_name);
124   printf("### Default action: %s\n", a->description.name);
125 }
126 void set_default_action_nat_dw()
127 {
128   char buffer[2048];
129   struct p4_header* h;
130   struct p4_set_default_action* sda;
131   struct p4_action* a;
132
133   h = create_p4_header(buffer, 0, sizeof(buffer));
134
135   sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
136         strcpy(sda->table_name, "nat_dw");
137
138   a = &(sda->action);
139   strcpy(a->description.name, "drop");
140
141   netconv_p4_header(h);
142   netconv_p4_set_default_action(sda);
143   netconv_p4_action(a);
144
145   send_p4_msg(c, buffer, sizeof(buffer));
146
147   printf("\n");
```

```
148   printf("Table name: %s\n", sda->table_name);
149   printf("### Default action: %s\n", a->description.name);
150 }
151
152 void set_default_action_meta_info()
153 {
154   char buffer[2048];
155   struct p4_header* h;
156   struct p4_set_default_action* sda;
157   struct p4_action* a;
158
159   h = create_p4_header(buffer, 0, sizeof(buffer));
160
161   sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
162         strcpy(sda->table_name, "meta_info");
163
164   a = &(sda->action);
165   strcpy(a->description.name, "set_meta_info");
166
167   netconv_p4_header(h);
168   netconv_p4_set_default_action(sda);
169   netconv_p4_action(a);
170
171   send_p4_msg(c, buffer, sizeof(buffer));
172
173   printf("\n");
174   printf("Table name: %s\n", sda->table_name);
175   printf("### Default action: %s\n", a->description.name);
176 }
177
178 void set_default_action_decap_process()
179 {
180   char buffer[2048];
181   struct p4_header* h;
182   struct p4_set_default_action* sda;
183   struct p4_action* a;
184
185   h = create_p4_header(buffer, 0, sizeof(buffer));
186
187   sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
188         strcpy(sda->table_name, "decap_process_outer");
189
190   a = &(sda->action);
191   strcpy(a->description.name, "drop");
192
193   netconv_p4_header(h);
194   netconv_p4_set_default_action(sda);
195   netconv_p4_action(a);
196
197   send_p4_msg(c, buffer, sizeof(buffer));
198
199   printf("\n");
200   printf("Table name: %s\n", sda->table_name);
201   printf("### Default action: %s\n", a->description.name);
202 }
203
204
205 void set_default_action_encap_process()
206 {
207   char buffer[2048];
208   struct p4_header* h;
209   struct p4_set_default_action* sda;
210   struct p4_action* a;
211
212   h = create_p4_header(buffer, 0, sizeof(buffer));
213
214   sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
215         strcpy(sda->table_name, "tunnel_encap_process_outer");
216
217   a = &(sda->action);
218   strcpy(a->description.name, "drop");
219
220   netconv_p4_header(h);
221   netconv_p4_set_default_action(sda);
222   netconv_p4_action(a);
223
224   send_p4_msg(c, buffer, sizeof(buffer));
225
226   printf("\n");
227   printf("Table name: %s\n", sda->table_name);
228   printf("### Default action: %s\n", a->description.name);
229 }
230
231
232 void fill_if_info(uint8_t if_info, uint8_t is_ext  )
233 {
234     char buffer[2048];
235     struct p4_header* h;
236     struct p4_add_table_entry* te;
237
```

```
238     struct p4_action* a;
239     struct p4_action_parameter* ap;
240 
241     struct p4_field_match_exact* exact;
242 
243     h = create_p4_header(buffer, 0, 2048);
244     te = create_p4_add_table_entry(buffer,0,2048);
245     strcpy(te->table_name, "if_info");
246 
247     exact = add_p4_field_match_exact(te, 2048);
248     strcpy(exact->header.name, "meta.routing_metadata.if_index"); // key
249     //strcpy(exact->header.name, "standard_metadata.ingress_port"); // key
250     exact->bitmap[0] = if_info;
251     exact->bitmap[1] = 0;
252     exact->length = 2*8+0;
253     a = add_p4_action(h, 2048);
254     strcpy(a->description.name, "set_if_info");
255 
256     ap = add_p4_action_parameter(h, a, 2048);
257   strcpy(ap->name, "is_ext");
258   ap->bitmap[0] = is_ext;
259   ap->bitmap[1] = 0;
260   ap->length = 2*8+0;
261 
262 
263     netconv_p4_header(h);
264     netconv_p4_add_table_entry(te);
265     netconv_p4_field_match_exact(exact);
266     netconv_p4_action(a);
267     netconv_p4_action_parameter(ap);
268 
269     send_p4_msg(c, buffer, 2048);
270 
271 }
272 
273 
274 void fill_smac(uint8_t mac[6] )
275 {
276     char buffer[2048];
277     struct p4_header* h;
278     struct p4_add_table_entry* te;
279     struct p4_action* a;
280 
281     struct p4_field_match_exact* exact;
282     printf("fill_smac table update \n");
283 
284     h = create_p4_header(buffer, 0, 2048);
285     te = create_p4_add_table_entry(buffer,0,2048);
286     strcpy(te->table_name, "smac");
287 
288 
289     exact = add_p4_field_match_exact(te, 2048);
290     strcpy(exact->header.name, "hdr.ethernet.srcAddr"); // key
291     memcpy(exact->bitmap, mac, 6);
292     exact->length = 6*8+0;
293 
294     a = add_p4_action(h, 2048);
295     strcpy(a->description.name, "generate_learn_notify");
296 
297     netconv_p4_header(h);
298     netconv_p4_add_table_entry(te);
299     netconv_p4_field_match_exact(exact);
300     netconv_p4_action(a);
301     send_p4_msg(c, buffer, 2048);
302     printf("########## fill smac table \n");
303     printf("\n");
304     printf("Table name: %s\n", te->table_name);
305     printf("Action: %s\n", a->description.name);
306 
307 }
308 
309 void fill_nat_up(uint8_t ip_inn[4], uint8_t ip[4], uint8_t srctcp)
310 {
311   char buffer[2048];
312   struct p4_header* h;
313         struct p4_add_table_entry* te;
314   struct p4_action* a;
315         struct p4_action_parameter* ap,* ap2;
316 
317         struct p4_field_match_exact* exact;
318 
319   h = create_p4_header(buffer, 0, 2048);
320         te = create_p4_add_table_entry(buffer,0,2048);
321         strcpy(te->table_name, "nat_up");
322 
323     exact = add_p4_field_match_exact(te, 2048);
324   strcpy(exact->header.name, "hdr.inner_ipv4.srcAddr");
325     memcpy(exact->bitmap, ip_inn, 4);
326     exact->length = 4*8+0;
327 
```

```
328         a = add_p4_action(h, 2048);
329   strcpy(a->description.name, "nat_hit_int_to_ext");
330
331         ap = add_p4_action_parameter(h, a, 2048);
332   strcpy(ap->name, "srcAddr");
333         memcpy(ap->bitmap, ip, 4);
334         ap->length = 4*8+0;
335
336         ap2 = add_p4_action_parameter(h, a, 2048);
337         strcpy(ap2->name, "srcPort");
338         ap2->bitmap[0] = srctcp;
339         ap2->bitmap[1] = 0;
340         ap2->length = 2*8+0;
341
342   netconv_p4_header(h);
343         netconv_p4_add_table_entry(te);
344   netconv_p4_field_match_exact(exact);
345   netconv_p4_action(a);
346   netconv_p4_action_parameter(ap);
347   netconv_p4_action_parameter(ap2);
348
349   send_p4_msg(c, buffer, 2048);
350
351   printf("########## \n");
352   printf("\n");
353   printf("Table name: %s\n", te->table_name);
354 }
355  void fill_nat_dw(uint8_t port, uint8_t ip[4], uint8_t dst_tcp[2])
356 {
357   char buffer[2048];
358   struct p4_header* h;
359         struct p4_add_table_entry* te;
360   struct p4_action* a;
361         struct p4_action_parameter* ap,* ap2;
362
363         struct p4_field_match_exact* exact;
364
365   h = create_p4_header(buffer, 0, 2048);
366         te = create_p4_add_table_entry(buffer,0,2048);
367         strcpy(te->table_name, "nat_dw");
368
369         printf("Filling Table name: %s", te->table_name);
370
371         exact = add_p4_field_match_exact(te, 2048);
372   strcpy(exact->header.name, "meta.routing_metadata.is_ext_if");
373         exact->bitmap[0] = port;
374   exact->bitmap[1] = 0;
375   exact->length = 1*8+0;
376         printf("Table match: %s -> ", exact->bitmap);
377
378         a = add_p4_action(h, 2048);
379   strcpy(a->description.name, "nat_hit_ext_to_int");
380
381         printf("action: %s -> ", a->description.name );
382
383         ap = add_p4_action_parameter(h, a, 2048);
384   strcpy(ap->name, "dstAddr");
385         memcpy(ap->bitmap, ip, 4);
386         ap->length = 4*8+0;
387
388         printf("%s ->", ap->name);
389   for(int i = 0; i < 4; i++){
390                 printf("%d.",ap->bitmap[i]);
391         }
392
393
394   ap2 = add_p4_action_parameter(h, a, 2048);
395   strcpy(ap2->name, "dstPort");
396   memcpy(ap2->bitmap, dst_tcp, 2);
397   ap2->length = 2*8+0;
398
399         printf("%s ->", ap2->name);
400   for(int i = 0; i < 4; i++){
401                 printf("%d.",ap2->bitmap[i]);
402         }
403   printf("\n");
404   netconv_p4_header(h);
405         netconv_p4_add_table_entry(te);
406   netconv_p4_field_match_exact(exact);
407   netconv_p4_action(a);
408   netconv_p4_action_parameter(ap);
409   netconv_p4_action_parameter(ap2);
410
411   send_p4_msg(c, buffer, 2048);
412
413   printf("########## \n");
414   printf("\n");
415   printf("Table name: %s\n", te->table_name);
416   printf("Action: %s\n", a->description.name);
417 }
```

```
418
419 void fill_ipv4_lpm_dw(uint8_t dst_ip[4], uint8_t port, uint8_t nhop [4])
420 {
421   char buffer[2048];
422   struct p4_header* h;
423   struct p4_add_table_entry* te;
424   struct p4_action* a;
425   struct p4_action_parameter* ap,* ap2;
426   struct p4_field_match_exact* exact;
427
428   h = create_p4_header(buffer, 0, 2048);
429   te = create_p4_add_table_entry(buffer,0,2048);
430   strcpy(te->table_name, "ipv4_dw");
431
432   exact = add_p4_field_match_exact(te, 2048);
433   strcpy(exact->header.name, "hdr.outer_ipv4.dstAddr");
434   memcpy(exact->bitmap, dst_ip, 4);
435   exact->length = 4*8+0;
436
437   a = add_p4_action(h, 2048);
438   strcpy(a->description.name, "set_nhop");
439
440   ap = add_p4_action_parameter(h, a, 2048);
441   strcpy(ap->name, "port");
442   ap->bitmap[0] = port;
443   ap->bitmap[1] = 0;
444   ap->length = 2*8+0;
445
446         ap2 = add_p4_action_parameter(h, a, 2048);
447   strcpy(ap->name, "nhop_ipv4");
448         memcpy(ap->bitmap, nhop, 4);
449         ap->length = 4*8+0;
450
451
452   netconv_p4_header(h);
453   netconv_p4_add_table_entry(te);
454   netconv_p4_field_match_exact(exact);
455   netconv_p4_action(a);
456   netconv_p4_action_parameter(ap);
457   netconv_p4_action_parameter(ap2);
458   send_p4_msg(c, buffer, 2048);
459
460   printf("##########\n");
461         printf("Table name: %s\n", te->table_name);
462         printf("Match: %s\n", exact->header.name);
463         printf("Action: %s\n", a->description.name);
464         printf("Parameters: %s -> %d\n", ap->name, ap->bitmap[0]);
465 }
466 void fill_ipv4_lpm_up(uint8_t dst_ip[4], uint8_t port, uint8_t nhop [4])
467 {
468   char buffer[2048];
469   struct p4_header* h;
470   struct p4_add_table_entry* te;
471   struct p4_action* a;
472   struct p4_action_parameter* ap,* ap2;
473   struct p4_field_match_exact* exact;
474
475   h = create_p4_header(buffer, 0, 2048);
476   te = create_p4_add_table_entry(buffer,0,2048);
477   strcpy(te->table_name, "ipv4_up");
478
479   exact = add_p4_field_match_exact(te, 2048);
480   //strcpy(exact->header.name, "meta.routing_metadata.dst_ipv4");
481   strcpy(exact->header.name, "hdr.inner_ipv4.dstAddr");
482   memcpy(exact->bitmap, dst_ip, 4);
483   exact->length = 4*8+0;
484
485   a = add_p4_action(h, 2048);
486   strcpy(a->description.name, "set_nhop");
487
488   ap = add_p4_action_parameter(h, a, 2048);
489   strcpy(ap->name, "port");
490   ap->bitmap[0] = port;
491   ap->bitmap[1] = 0;
492   ap->length = 2*8+0;
493
494         ap2 = add_p4_action_parameter(h, a, 2048);
495   strcpy(ap->name, "nhop_ipv4");
496         memcpy(ap->bitmap, nhop, 4);
497         ap->length = 4*8+0;
498
499
500   netconv_p4_header(h);
501   netconv_p4_add_table_entry(te);
502   netconv_p4_field_match_exact(exact);
503   netconv_p4_action(a);
504   netconv_p4_action_parameter(ap);
505   netconv_p4_action_parameter(ap2);
506   send_p4_msg(c, buffer, 2048);
507
```

```
  printf("##########\n");
        printf("Table name: %s\n", te->table_name);
        printf("Match: %s\n", exact->header.name);
        printf("Action: %s\n", a->description.name);
        printf("Parameters: %s -> %d\n", ap->name, ap->bitmap[0]);
}

void fill_decap(uint8_t smac[6], uint8_t tn_id)
{
    char buffer[2048];
    struct p4_header* h;
    struct p4_add_table_entry* te;
    struct p4_action* a;
    struct p4_action_parameter* ap;

    struct p4_field_match_exact* exact;
    printf("Set decap entry... \n");

    h = create_p4_header(buffer, 0, 2048);
    te = create_p4_add_table_entry(buffer,0,2048);
    strcpy(te->table_name, "decap_process_outer"); // pilas


    exact = add_p4_field_match_exact(te, 2048);
    strcpy(exact->header.name, "hdr.ethernet.srcAddr"); // key
    memcpy(exact->bitmap, smac, 6);
    exact->length = 6*8+0;
    a = add_p4_action(h, 2048);
    strcpy(a->description.name, "decap_gre_inner_ipv4");

    ap = add_p4_action_parameter(h, a, 2048);
    strcpy(ap->name, "tunnel_id");
    ap->bitmap[0] = tn_id;
    ap->bitmap[1] = 0;
    ap->length = 2*8+0;


    netconv_p4_header(h);
    netconv_p4_add_table_entry(te);
    netconv_p4_field_match_exact(exact);
    netconv_p4_action(a);
    netconv_p4_action_parameter(ap);

    send_p4_msg(c, buffer, 2048);
    printf("##########\n");
    printf("\n");
    printf("Table name: %s\n", te->table_name);
}

void fill_encap(uint8_t dst_ip[4], uint8_t src_gre[4])
{
    char buffer[2048];
    struct p4_header* h;
    struct p4_add_table_entry* te;
    struct p4_action* a;
    struct p4_action_parameter* ap;
    struct p4_field_match_exact* exact;

    h = create_p4_header(buffer, 0, 2048);
    te = create_p4_add_table_entry(buffer,0,2048);
    strcpy(te->table_name, "tunnel_encap_process_outer");

    exact = add_p4_field_match_exact(te, 2048);
    strcpy(exact->header.name, "hdr.ipv4.dstAddr"); // key
    memcpy(exact->bitmap, dst_ip, 4);
    exact->length = 4*8+0;


    a = add_p4_action(h, 2048);
    strcpy(a->description.name, "ipv4_gre_rewrite");

    ap = add_p4_action_parameter(h, a, 2048);
    strcpy(ap->name, "gre_srcAddr");
    memcpy(ap->bitmap, src_gre, 4);
    ap->length = 4*8+0;


    netconv_p4_header(h);
    netconv_p4_add_table_entry(te);
    netconv_p4_field_match_exact(exact);
    netconv_p4_action(a);
    netconv_p4_action_parameter(ap);

    send_p4_msg(c, buffer, 2048);
    printf("##########\n");
    printf("\n");
    printf("Table name: %s\n", te->table_name);
    printf("Match: %s\n", exact->header.name);
    printf("Action: %s\n", a->description.name);
    printf("Parameters: %s -> %d\n", ap->name, ap->bitmap[0]);
```

```
598 }
599 void fill_sendout_table(uint8_t port, uint8_t smac[6])
600 {
601     char buffer[2048];
602     struct p4_header* h;
603     struct p4_add_table_entry* te;
604     struct p4_action* a;
605     struct p4_action_parameter* ap;
606     struct p4_field_match_exact* exact;
607 
608     h = create_p4_header(buffer, 0, 2048);
609     te = create_p4_add_table_entry(buffer,0,2048);
610     strcpy(te->table_name, "sendout");
611 
612     exact = add_p4_field_match_exact(te, 2048);
613     strcpy(exact->header.name, "standard_metadata.egress_port");
614     exact->bitmap[0] = port;
615     exact->bitmap[1] = 0;
616     exact->length = 2*8+0;
617 
618   a = add_p4_action(h, 2048);
619   strcpy(a->description.name, "rewrite_src_mac");
620 
621   ap = add_p4_action_parameter(h, a, 2048);
622   strcpy(ap->name, "src_mac");
623   memcpy(ap->bitmap, smac, 6);
624   ap->length = 6*8+0;
625 
626     netconv_p4_header(h);
627     netconv_p4_add_table_entry(te);
628     netconv_p4_field_match_exact(exact);
629     netconv_p4_action(a);
630     netconv_p4_action_parameter(ap);
631     send_p4_msg(c, buffer, 2048);
632 
633   printf("##########\n");
634         printf("\n");
635         printf("Table name: %s\n", te->table_name);
636   printf("Match: %s\n", exact->header.name);
637         printf("Action: %s\n", a->description.name);
638   printf("Parameters: %s ->", ap->name);
639 
640   for(int i = 0; i < 6; i++){
641     printf("%d",ap->bitmap[i]);
642   }
643 
644 
645 }
646 
647 void fill_sendout_dw_table(uint8_t port, uint8_t smac[6])
648 {
649     char buffer[2048];
650     struct p4_header* h;
651     struct p4_add_table_entry* te;
652     struct p4_action* a;
653     struct p4_action_parameter* ap;
654     struct p4_field_match_exact* exact;
655 
656     h = create_p4_header(buffer, 0, 2048);
657     te = create_p4_add_table_entry(buffer,0,2048);
658     strcpy(te->table_name, "sendout_dw");
659 
660     exact = add_p4_field_match_exact(te, 2048);
661     strcpy(exact->header.name, "standard_metadata.egress_port");
662     exact->bitmap[0] = port;
663     exact->length = 1*8+0;
664 
665   a = add_p4_action(h, 2048);
666   strcpy(a->description.name, "rewrite_src_mac_dw");
667 
668   ap = add_p4_action_parameter(h, a, 2048);
669   strcpy(ap->name, "src_mac");
670   memcpy(ap->bitmap, smac, 6);
671   ap->length = 6*8+0;
672 
673     netconv_p4_header(h);
674     netconv_p4_add_table_entry(te);
675     netconv_p4_field_match_exact(exact);
676     netconv_p4_action(a);
677     netconv_p4_action_parameter(ap);
678     send_p4_msg(c, buffer, 2048);
679 
680   printf("##########\n");
681         printf("\n");
682         printf("Table name: %s\n", te->table_name);
683   printf("Match: %s\n", exact->header.name);
684         printf("Action: %s\n", a->description.name);
685         printf("key egress_port: %d",exact->bitmap[0]);
686 
687 }
```

```
688
689
690 void dhf(void* b) {
691   printf("Unknown digest received\n");
692 }
693
694 void init() {
695          printf("Set default actions.\n");
696
697   int i;
698   uint8_t port1 = 1;
699   uint8_t port0 = 0;
700   uint8_t tn_id0 = 100;
701
702   uint8_t ip2[4] = {192,168,0,1};
703   uint8_t stcp[2] = {10};
704         uint8_t mac_if1[6] = {0xaa, 0xbb, 0xcc, 0xaa, 0xdd, 0xee};
705
706         uint8_t is_ext = 1;
707         uint8_t is_int = 0;
708
709         set_default_action_decap_process();
710         set_default_action_encap_process();
711         set_default_action_nat_up();
712
713         fill_if_info(port0,is_int);
714         fill_sendout_table(port1, mac_if1);
715
716         for (i=0;i<=mac_count;++i)
717         {
718
719                 printf("Filling tables smac/decap/nat_up/lpm_up MAC: %02x:%02x:%02x:%02x:%02x:%02x src_IP: %d.%d.%d.%d
                        dst_IP: %d.%d.%d.%d srcTCP:%d  d\n ", macs[i][0],macs[i][1],macs[i][2],macs[i][3],macs[i][4],macs
                        [i][5], ips[i][0],ips[i][1],ips[i][2],ips[i][3] , ipd[i][0],ipd[i][1],ipd[i][2],ipd[i][3]),
                        stcp_txt[i][1];
720
721
722                fill_smac(macs[i]);
723                fill_decap(macs[i], tn_id0);
724                fill_nat_up(ips[i],ips_new[i],stcp_txt[i][1]);
725                fill_ipv4_lpm_up(ipd[i],port1 , ipd[i]);
726
727                if(0 == (i%500)){ printf("inside sleep \n");sleep(1);;}
728
729
730                usleep(1000);
731         }
732
733         printf ("ctrl Total entries sent %d\n",i);
734         printf("\n");
735
736  }
737
738  int main(int argc, char* argv[])
739  {
740          if (argc>1) {
741                  if (argc!=2) {
742                          printf("Too many arguments...\nUsage: %s <filename(optional)>\n", argv[0]);
743                          return -1;
744                  }
745                  printf("Command line argument is present...\nLoading configuration data...\n");
746                  if (read_macs_and_ports_from_file(argv[1])<0) {
747                          printf("File cannnot be opened...\n");
748                          return -1;
749                  }
750          }
751
752          printf("Create and configure controller...\n");
753          c = create_controller_with_init(11111, 3, dhf, init);
754          printf("MACSAD controller started...\n");
755          execute_controller(c);
756
757          printf("MACSAD controller terminated\n");
758          destroy_controller(c);
759          return 0;
760  }
```

BNG controller for DL

```
1  #include "controller.h"
2  #include "messages.h"
3  #include <unistd.h>
4  #include <stdio.h>
5  #include <string.h>
6
7  #define MAX_MACS 1000000
8
9  controller c;
10
```

```
11 uint8_t macs[MAX_MACS][6];
12 uint8_t portmap[MAX_MACS];
13 uint8_t ipd_gre[MAX_MACS][4];
14 uint8_t ipd[MAX_MACS][4];
15 uint8_t dtcp_txt[MAX_MACS][2];
16 uint8_t dtcp_new_txt[MAX_MACS][2];
17
18 int mac_count = -1;
19
20 int read_macs_and_ports_from_file(char *filename) {
21   FILE *f;
22   char line[200];
23   int values[6];
24   int values_ip[4];
25   int values_ip2[4];
26   int values_dtcp[2];
27   int values_dtcp_nw[2];
28         int port;
29   int i;
30
31   f = fopen(filename, "r");
32   if (f == NULL) return -1;
33
34   while (fgets(line, sizeof(line), f)) {
35     line[strlen(line)-1] = '\0';
36     if (16 == sscanf(line, "%x:%x:%x:%x:%x:%x %d.%d.%d.%d %d.%d.%d.%d %d %d",
37           &values[0], &values[1], &values[2],
38           &values[3], &values[4], &values[5],
39           &values_ip[0], &values_ip[1], &values_ip[2], &values_ip[3],
40           &values_ip2[0], &values_ip2[1], &values_ip2[2], &values_ip2[3],
41           &values_dtcp[1],
42           &port
43                 ) )
44     {
45       if (mac_count==MAX_MACS-1)
46       {
47         printf("Too many entries...\n");
48         break;
49       }
50
51       ++mac_count;
52       for( i = 0; i < 6; ++i )
53         macs[mac_count][i] = (uint8_t) values[i];
54       for( i = 0; i < 4; ++i )
55           ipd[mac_count][i] = (uint8_t) values_ip[i];
56         for( i = 0; i < 4; ++i )
57
58         ipd_gre[mac_count][i] = (uint8_t) values_ip2[i];
59           dtcp_txt[mac_count][1] = (uint8_t) values_dtcp[1];
60           dtcp_txt[mac_count][0] = 0;
61           portmap[mac_count] = (uint8_t) port;
62
63
64       } else {
65         printf("Wrong format error in line %d : %s\n", mac_count+2, line);
66         fclose(f);
67         return -1;
68       }
69
70     }
71
72     fclose(f);
73     return 0;
74   }
75
76
77   void set_default_action_ipv4_forward()
78   {
79     char buffer[2048];
80     struct p4_header* h;
81     struct p4_set_default_action* sda;
82     struct p4_action* a;
83
84     h = create_p4_header(buffer, 0, sizeof(buffer));
85     sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
86     strcpy(sda->table_name, "ipv4_forward");
87
88     a = &(sda->action);
89     strcpy(a->description.name, "drop");
90
91     netconv_p4_header(h);
92     netconv_p4_set_default_action(sda);
93     netconv_p4_action(a);
94     send_p4_msg(c, buffer, sizeof(buffer));
95
96   }
97   void set_default_action_nat_up()
98   {
99     char buffer[2048];
100    struct p4_header* h;
```

```
101     struct p4_set_default_action* sda;
102     struct p4_action* a;
103
104     h = create_p4_header(buffer, 0, sizeof(buffer));
105
106     sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
107     strcpy(sda->table_name, "nat_up");
108
109     a = &(sda->action);
110     strcpy(a->description.name, "drop");
111     netconv_p4_header(h);
112     netconv_p4_set_default_action(sda);
113     netconv_p4_action(a);
114     send_p4_msg(c, buffer, sizeof(buffer));
115
116   }
117   void set_default_action_nat_dw()
118   {
119     char buffer[2048];
120     struct p4_header* h;
121     struct p4_set_default_action* sda;
122     struct p4_action* a;
123
124     h = create_p4_header(buffer, 0, sizeof(buffer));
125
126     sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
127     strcpy(sda->table_name, "nat_dw");
128
129     a = &(sda->action);
130     strcpy(a->description.name, "drop");
131
132     netconv_p4_header(h);
133     netconv_p4_set_default_action(sda);
134     netconv_p4_action(a);
135
136     send_p4_msg(c, buffer, sizeof(buffer));
137
138   }
139
140   void set_default_action_meta_info()
141   {
142     char buffer[2048];
143     struct p4_header* h;
144     struct p4_set_default_action* sda;
145     struct p4_action* a;
146
147     h = create_p4_header(buffer, 0, sizeof(buffer));
148
149     sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
150     strcpy(sda->table_name, "meta_info");
151
152     a = &(sda->action);
153     strcpy(a->description.name, "set_meta_info");
154
155     netconv_p4_header(h);
156     netconv_p4_set_default_action(sda);
157     netconv_p4_action(a);
158
159     send_p4_msg(c, buffer, sizeof(buffer));
160
161   }
162
163   void set_default_action_decap_process()
164   {
165     char buffer[2048];
166     struct p4_header* h;
167     struct p4_set_default_action* sda;
168     struct p4_action* a;
169
170     h = create_p4_header(buffer, 0, sizeof(buffer));
171
172     sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
173     strcpy(sda->table_name, "decap_process_outer");
174
175     a = &(sda->action);
176     strcpy(a->description.name, "drop");
177
178     netconv_p4_header(h);
179     netconv_p4_set_default_action(sda);
180     netconv_p4_action(a);
181
182     send_p4_msg(c, buffer, sizeof(buffer));
183
184   }
185
186
187   void set_default_action_encap_process()
188   {
189     char buffer[2048];
190     struct p4_header* h;
```

```
191     struct p4_set_default_action* sda;
192     struct p4_action* a;
193
194     h = create_p4_header(buffer, 0, sizeof(buffer));
195
196     sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
197     strcpy(sda->table_name, "tunnel_encap_process_outer");
198
199     a = &(sda->action);
200     strcpy(a->description.name, "drop");
201
202     netconv_p4_header(h);
203     netconv_p4_set_default_action(sda);
204     netconv_p4_action(a);
205
206     send_p4_msg(c, buffer, sizeof(buffer));
207
208   }
209
210   void set_default_action_firewall_dl()
211   {
212     char buffer[2048];
213     struct p4_header* h;
214     struct p4_set_default_action* sda;
215     struct p4_action* a;
216
217     h = create_p4_header(buffer, 0, sizeof(buffer));
218
219     sda = create_p4_set_default_action(buffer,0,sizeof(buffer));
220     strcpy(sda->table_name, "firewall_dl");
221
222     a = &(sda->action);
223     strcpy(a->description.name, "nop");
224
225     netconv_p4_header(h);
226     netconv_p4_set_default_action(sda);
227     netconv_p4_action(a);
228
229     send_p4_msg(c, buffer, sizeof(buffer));
230
231   }
232
233   void fill_if_info(uint8_t if_info, uint8_t is_ext  )
234   {
235       char buffer[2048];
236       struct p4_header* h;
237       struct p4_add_table_entry* te;
238
239       struct p4_action* a;
240       struct p4_action_parameter* ap;
241
242       struct p4_field_match_exact* exact;
243
244       h = create_p4_header(buffer, 0, 2048);
245       te = create_p4_add_table_entry(buffer,0,2048);
246       strcpy(te->table_name, "if_info");
247
248       exact = add_p4_field_match_exact(te, 2048);
249       strcpy(exact->header.name, "meta.routing_metadata.if_index"); // key
250       exact->bitmap[0] = if_info;
251       exact->bitmap[1] = 0;
252       exact->length = 2*8+0;
253       a = add_p4_action(h, 2048);
254       strcpy(a->description.name, "set_if_info");
255
256     ap = add_p4_action_parameter(h, a, 2048);
257     strcpy(ap->name, "is_ext");
258     ap->bitmap[0] = is_ext;
259     ap->bitmap[1] = 0;
260     ap->length = 2*8+0;
261
262
263       netconv_p4_header(h);
264       netconv_p4_add_table_entry(te);
265       netconv_p4_field_match_exact(exact);
266       netconv_p4_action(a);
267       netconv_p4_action_parameter(ap);
268
269       send_p4_msg(c, buffer, 2048);
270
271   }
272
273
274   void fill_smac(uint8_t mac[6] )
275   {
276       char buffer[2048];
277       struct p4_header* h;
278       struct p4_add_table_entry* te;
279       struct p4_action* a;
280
```

```
281       struct p4_field_match_exact* exact;
282
283       h = create_p4_header(buffer, 0, 2048);
284       te = create_p4_add_table_entry(buffer,0,2048);
285       strcpy(te->table_name, "smac");
286
287
288       exact = add_p4_field_match_exact(te, 2048);
289       strcpy(exact->header.name, "hdr.ethernet.srcAddr"); // key
290       memcpy(exact->bitmap, mac, 6);
291       exact->length = 6*8+0;
292
293       a = add_p4_action(h, 2048);
294       strcpy(a->description.name, "generate_learn_notify");
295
296       netconv_p4_header(h);
297       netconv_p4_add_table_entry(te);
298       netconv_p4_field_match_exact(exact);
299       netconv_p4_action(a);
300       send_p4_msg(c, buffer, 2048);
301
302   }
303
304   void fill_nat_up(uint8_t ip_inn[4], uint8_t ip[4], uint8_t srctcp[2])
305   {
306     char buffer[2048];
307     struct p4_header* h;
308     struct p4_add_table_entry* te;
309     struct p4_action* a;
310     struct p4_action_parameter* ap,* ap2;
311
312     struct p4_field_match_exact* exact;
313
314     h = create_p4_header(buffer, 0, 2048);
315     te = create_p4_add_table_entry(buffer,0,2048);
316     strcpy(te->table_name, "nat_up");
317
318          // printf("Fill Table name: %s", te->table_name);
319
320     exact = add_p4_field_match_exact(te, 2048);
321     strcpy(exact->header.name, "hdr.inner_ipv4.srcAddr");
322     memcpy(exact->bitmap, ip_inn, 4);
323     exact->length = 4*8+0;
324     a = add_p4_action(h, 2048);
325     strcpy(a->description.name, "nat_hit_int_to_ext");
326
327     ap = add_p4_action_parameter(h, a, 2048);
328     strcpy(ap->name, "srcAddr");
329     memcpy(ap->bitmap, ip, 4);
330     ap->length = 4*8+0;
331
332     ap2 = add_p4_action_parameter(h, a, 2048);
333     strcpy(ap2->name, "srcPort");
334     memcpy(ap2->bitmap, srctcp, 2);
335     ap2->length = 2*8+0;
336
337
338     netconv_p4_header(h);
339     netconv_p4_add_table_entry(te);
340     netconv_p4_field_match_exact(exact);
341     netconv_p4_action(a);
342     netconv_p4_action_parameter(ap);
343     netconv_p4_action_parameter(ap2);
344
345
346
347     send_p4_msg(c, buffer, 2048);
348
349   }
350   void fill_nat_dw(uint8_t dtcp0, uint8_t ip[4])
351   {
352     char buffer[2048];
353     struct p4_header* h;
354     struct p4_add_table_entry* te;
355     struct p4_action* a;
356     struct p4_action_parameter* ap,* ap2;
357
358     struct p4_field_match_exact* exact;
359
360     h = create_p4_header(buffer, 0, 2048);
361     te = create_p4_add_table_entry(buffer,0,2048);
362     strcpy(te->table_name, "nat_dw");
363
364
365     exact = add_p4_field_match_exact(te, 2048);
366     strcpy(exact->header.name, "hdr.tcp.dstPort");
367     exact->bitmap[1] = dtcp0;
368     exact->bitmap[0] = 0;
369     exact->length = 2*8+0;
370
```

```
371     printf("*/*/*/*/*%s.",exact->bitmap);
372     a = add_p4_action(h, 2048);
373     strcpy(a->description.name, "nat_hit_ext_to_int");
374
375     ap = add_p4_action_parameter(h, a, 2048);
376     strcpy(ap->name, "dstAddr");
377     memcpy(ap->bitmap, ip, 4);
378     ap->length = 4*8+0;
379
380
381
382     printf("#### Filling DW : %s", te->table_name);
383     netconv_p4_header(h);
384     netconv_p4_add_table_entry(te);
385     netconv_p4_field_match_exact(exact);
386     netconv_p4_action(a);
387     netconv_p4_action_parameter(ap);
388     netconv_p4_action_parameter(ap2);
389
390     send_p4_msg(c, buffer, 2048);
391     printf("\n");
392
393   }
394
395   void fill_ipv4_lpm_dw(uint8_t dst_ip[4], uint8_t port, uint8_t nhop [4])
396   {
397     char buffer[2048];
398     struct p4_header* h;
399     struct p4_add_table_entry* te;
400     struct p4_action* a;
401     struct p4_action_parameter* ap,* ap2;
402     struct p4_field_match_exact* exact;
403
404     h = create_p4_header(buffer, 0, 2048);
405     te = create_p4_add_table_entry(buffer,0,2048);
406     strcpy(te->table_name, "ipv4_dw");
407
408     exact = add_p4_field_match_exact(te, 2048);
409     strcpy(exact->header.name, "hdr.outer_ipv4.dstAddr");
410     memcpy(exact->bitmap, dst_ip, 4);
411     exact->length = 4*8+0;
412
413     a = add_p4_action(h, 2048);
414     strcpy(a->description.name, "set_nhop");
415
416     ap = add_p4_action_parameter(h, a, 2048);
417     strcpy(ap->name, "port");
418     ap->bitmap[0] = port;
419     ap->bitmap[1] = 0;
420     ap->length = 2*8+0;
421
422     ap2 = add_p4_action_parameter(h, a, 2048);
423     strcpy(ap->name, "nhop_ipv4");
424     memcpy(ap->bitmap, nhop, 4);
425     ap->length = 4*8+0;
426
427
428     netconv_p4_header(h);
429     netconv_p4_add_table_entry(te);
430     netconv_p4_field_match_exact(exact);
431     netconv_p4_action(a);
432     netconv_p4_action_parameter(ap);
433     netconv_p4_action_parameter(ap2);
434     send_p4_msg(c, buffer, 2048);
435
436   }
437   void fill_ipv4_lpm_up(uint8_t dst_ip[4], uint8_t port, uint8_t nhop [4])
438   {
439     char buffer[2048];
440     struct p4_header* h;
441     struct p4_add_table_entry* te;
442     struct p4_action* a;
443     struct p4_action_parameter* ap,* ap2;
444     struct p4_field_match_exact* exact;
445
446     h = create_p4_header(buffer, 0, 2048);
447     te = create_p4_add_table_entry(buffer,0,2048);
448     strcpy(te->table_name, "ipv4_up");
449
450     exact = add_p4_field_match_exact(te, 2048);
451     //strcpy(exact->header.name, "meta.routing_metadata.dst_ipv4");
452     strcpy(exact->header.name, "hdr.inner_ipv4.dstAddr");
453     memcpy(exact->bitmap, dst_ip, 4);
454     exact->length = 4*8+0;
455
456     a = add_p4_action(h, 2048);
457     strcpy(a->description.name, "set_nhop");
458
459     ap = add_p4_action_parameter(h, a, 2048);
460     strcpy(ap->name, "port");
```

```
461     ap->bitmap[0] = port;
462     ap->bitmap[1] = 0;
463     ap->length = 2*8+0;
464
465     ap2 = add_p4_action_parameter(h, a, 2048);
466     strcpy(ap->name, "nhop_ipv4");
467     memcpy(ap->bitmap, nhop, 4);
468     ap->length = 4*8+0;
469
470
471     netconv_p4_header(h);
472     netconv_p4_add_table_entry(te);
473     netconv_p4_field_match_exact(exact);
474     netconv_p4_action(a);
475     netconv_p4_action_parameter(ap);
476     netconv_p4_action_parameter(ap2);
477     send_p4_msg(c, buffer, 2048);
478
479   }
480
481   void fill_decap(uint8_t smac[6], uint8_t tn_id)
482   {
483       char buffer[2048];
484       struct p4_header* h;
485       struct p4_add_table_entry* te;
486       struct p4_action* a;
487       struct p4_action_parameter* ap;
488
489       struct p4_field_match_exact* exact;
490
491       h = create_p4_header(buffer, 0, 2048);
492       te = create_p4_add_table_entry(buffer,0,2048);
493       strcpy(te->table_name, "decap_process_outer");
494
495
496       exact = add_p4_field_match_exact(te, 2048);
497       strcpy(exact->header.name, "hdr.ethernet.srcAddr"); // key
498       memcpy(exact->bitmap, smac, 6);
499       exact->length = 6*8+0;
500
501       a = add_p4_action(h, 2048);
502       strcpy(a->description.name, "decap_gre_inner_ipv4");
503
504       ap = add_p4_action_parameter(h, a, 2048);
505       strcpy(ap->name, "tunnel_id");
506       ap->bitmap[0] = tn_id;
507       ap->bitmap[1] = 0;
508       ap->length = 2*8+0;
509
510
511       netconv_p4_header(h);
512       netconv_p4_add_table_entry(te);
513       netconv_p4_field_match_exact(exact);
514       netconv_p4_action(a);
515       netconv_p4_action_parameter(ap);
516
517       send_p4_msg(c, buffer, 2048);
518   }
519   // ------------------------------------------------------------------------
520
521   void fill_encap(uint8_t dst_ip[4], uint8_t src_gre[4])
522   {
523       char buffer[2048];
524       struct p4_header* h;
525       struct p4_add_table_entry* te;
526       struct p4_action* a;
527       struct p4_action_parameter* ap;
528       struct p4_field_match_exact* exact;
529
530       h = create_p4_header(buffer, 0, 2048);
531       te = create_p4_add_table_entry(buffer,0,2048);
532       strcpy(te->table_name, "tunnel_encap_process_outer");
533
534       exact = add_p4_field_match_exact(te, 2048);
535       strcpy(exact->header.name, "hdr.ipv4.dstAddr"); // key
536       memcpy(exact->bitmap, dst_ip, 4);
537       exact->length = 4*8+0;
538
539
540       a = add_p4_action(h, 2048);
541       strcpy(a->description.name, "ipv4_gre_rewrite");
542
543       ap = add_p4_action_parameter(h, a, 2048);
544       strcpy(ap->name, "gre_dstAddr");
545       memcpy(ap->bitmap, src_gre, 4);
546       ap->length = 4*8+0;
547
548       netconv_p4_header(h);
549       netconv_p4_add_table_entry(te);
550       netconv_p4_field_match_exact(exact);
```

```
551       netconv_p4_action(a);
552       netconv_p4_action_parameter(ap);
553
554       send_p4_msg(c, buffer, 2048);
555   }
556   void fill_sendout_table(uint8_t port, uint8_t smac[6])
557   {
558       char buffer[2048];
559       struct p4_header* h;
560       struct p4_add_table_entry* te;
561       struct p4_action* a;
562       struct p4_action_parameter* ap;
563       struct p4_field_match_exact* exact;
564
565       h = create_p4_header(buffer, 0, 2048);
566       te = create_p4_add_table_entry(buffer,0,2048);
567       strcpy(te->table_name, "sendout");
568
569       exact = add_p4_field_match_exact(te, 2048);
570       strcpy(exact->header.name, "standard_metadata.egress_port");
571       exact->bitmap[0] = port;
572       exact->bitmap[1] = 0;
573       exact->length = 2*8+0;
574
575     a = add_p4_action(h, 2048);
576     strcpy(a->description.name, "rewrite_src_mac");
577
578     ap = add_p4_action_parameter(h, a, 2048);
579     strcpy(ap->name, "src_mac");
580     memcpy(ap->bitmap, smac, 6);
581     ap->length = 6*8+0;
582
583       netconv_p4_header(h);
584       netconv_p4_add_table_entry(te);
585       netconv_p4_field_match_exact(exact);
586       netconv_p4_action(a);
587       netconv_p4_action_parameter(ap);
588       send_p4_msg(c, buffer, 2048);
589
590   }
591
592   void fill_sendout_dw_table(uint8_t port, uint8_t smac[6])
593   {
594       char buffer[2048];
595       struct p4_header* h;
596       struct p4_add_table_entry* te;
597       struct p4_action* a;
598       struct p4_action_parameter* ap;
599       struct p4_field_match_exact* exact;
600
601       h = create_p4_header(buffer, 0, 2048);
602       te = create_p4_add_table_entry(buffer,0,2048);
603       strcpy(te->table_name, "sendout_dw");
604
605       exact = add_p4_field_match_exact(te, 2048);
606       strcpy(exact->header.name, "standard_metadata.egress_port");
607       exact->bitmap[0] = port;
608       exact->length = 1*8+0;
609
610     a = add_p4_action(h, 2048);
611     strcpy(a->description.name, "rewrite_src_mac_dw");
612
613     ap = add_p4_action_parameter(h, a, 2048);
614     strcpy(ap->name, "src_mac");
615     memcpy(ap->bitmap, smac, 6);
616     ap->length = 6*8+0;
617
618       netconv_p4_header(h);
619       netconv_p4_add_table_entry(te);
620       netconv_p4_field_match_exact(exact);
621       netconv_p4_action(a);
622       netconv_p4_action_parameter(ap);
623       send_p4_msg(c, buffer, 2048);
624   }
625
626   void fill_firewall_dw(uint8_t dst_ip[4])
627   {
628     char buffer[2048];
629     struct p4_header* h;
630     struct p4_add_table_entry* te;
631     struct p4_action* a;
632     struct p4_action_parameter* ap,* ap2;
633     struct p4_field_match_exact* exact;
634
635     h = create_p4_header(buffer, 0, 2048);
636     te = create_p4_add_table_entry(buffer,0,2048);
637     strcpy(te->table_name, "firewall_dl");
638
639     exact = add_p4_field_match_exact(te, 2048);
640     strcpy(exact->header.name, "hdr.ipv4.dstAddr");
```

```
641     memcpy(exact->bitmap, dst_ip, 4);
642     exact->length = 4*8+0;
643 
644     a = add_p4_action(h, 2048);
645     strcpy(a->description.name, "drop");
646 
647 
648 
649     netconv_p4_header(h);
650     netconv_p4_add_table_entry(te);
651     netconv_p4_field_match_exact(exact);
652     netconv_p4_action(a);
653     send_p4_msg(c, buffer, 2048);
654 
655   }
656 
657   void dhf(void* b) {
658     printf("Unknown digest received\n");
659   }
660 
661   void init() {
662      printf("Set default actions.\n");
663 
664     int i;
665     uint8_t port1 = 1;
666     uint8_t port0 = 0;
667     uint8_t tn_id0 = 100;
668 
669     uint8_t ip2[4] = {192,168,0,1};
670 
671     uint8_t end_gre[4] = {11,0,0,10};      //1
672     uint8_t ip_dst_up[4] = {192,168,0,10};
673     uint8_t ip_dst_dw[4] = {10,0,0,10};     //2
674     uint8_t stcp[2] = {10};
675     uint8_t dtcp = 102;   // 3
676     int8_t smac_2[6] = {0x00, 0x55, 0x00, 0x00, 0x00, 0x00};  //
677     uint8_t mac_if1[6] = {0xaa, 0xbb, 0xcc, 0xaa, 0xdd, 0xee};
678 
679     uint8_t is_ext = 1;
680     uint8_t is_int = 0;
681 
682     set_default_action_decap_process();
683     set_default_action_encap_process();
684     set_default_action_nat_up();
685     set_default_action_nat_dw();
686     set_default_action_firewall_dl();
687 
688 
689     fill_smac(smac_2);
690 
691     fill_sendout_dw_table(port1, mac_if1);
692 
693     //--------- test
694     fill_nat_dw(dtcp,ip_dst_dw);   //  ips this case is ip_dst_dw:10.0.0.10
695         fill_encap(ip_dst_dw,end_gre); //*DL
696         fill_ipv4_lpm_up(end_gre,port1,end_gre);
697 
698         for (i=0;i<=mac_count;++i)
699         {
700 
701                 printf("\n Filling tables smac/decap/nat_up/lpm_up MAC: %02x:%02x:%02x:%02x:%02x:%02x src_IP: %d.%d.%d.%
                        d dst_IP: %d.%d.%d.%d srcTcpPort %d port %d \n", macs[i][0],macs[i][1],macs[i][2],macs[i][3],macs[
                        i][4],macs[i][5], ipd[i][0],ipd[i][1],ipd[i][2],ipd[i][3] , ipd_gre[i][0],ipd_gre[i][1],ipd_gre[i
                        ][2],ipd_gre[i][3], dtcp_txt[i][1], portmap[i] );
702 
703                 fill_nat_dw(dtcp_txt[i][1], ipd[i]);
704                 fill_encap(ipd[i],ipd_gre[i]);
705                 fill_ipv4_lpm_up(ipd_gre[i],port1, ipd_gre[i]);
706 
707                 if(0 == (i%100)){ printf("inside sleep \n");sleep(1);;}
708                  usleep(10000);
709 
710         }
711   }
712   int main(int argc, char* argv[])
713   {
714           if (argc>1) {
715                   if (argc!=2) {
716                           printf("Too many arguments...\nUsage: %s <filename(optional)>\n", argv[0]);
717                           return -1;
718                   }
719                   printf("Command line argument is present...\nLoading configuration data...\n");
720                   if (read_macs_and_ports_from_file(argv[1])<0) {
721                           printf("File cannnot be opened...\n");
722                           return -1;
723                   }
724           }
725 
726           printf("Create and configure controller...\n");
727           c = create_controller_with_init(11111, 3, dhf, init);
```

```
728         printf("MACSAD controller started...\n");
729         execute_controller(c);
730
731         printf("MACSAD controller terminated\n");
732         destroy_controller(c);
733         return 0;
734 }
```

E.2 ODP application example: L2 simple forwarding

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <getopt.h>
4  #include <signal.h>
5
6  #include <odp_api.h>
7  #include <odp/helper/odph_api.h>
8
9  #define POOL_NUM_PKT 8192
10 #define POOL_SEG_LEN 1856
11 #define MAX_PKT_BURST 32
12 #define MAX_WORKERS 1
13
14 typedef struct {
15   odp_pktio_t if0, if1;
16   odp_pktin_queue_t if0in, if1in;
17   odp_pktout_queue_t if0out, if1out;
18   odph_ethaddr_t src, dst;
19   odp_shm_t shm;
20   int exit_thr;
21   int wait_sec;
22 } global_data_t;
23
24 static global_data_t *global;
25
26 static void sig_handler(int signo ODP_UNUSED)
27 {
28   printf("sig_handler!\n");
29   global->exit_thr = 1;
30 }
31
32 static odp_pktio_t create_pktio(const char *name, odp_pool_t pool,
33         odp_pktin_queue_t *pktin,
34         odp_pktout_queue_t *pktout)
35 {
36   odp_pktio_param_t pktio_param;
37   odp_pktin_queue_param_t in_queue_param;
38   odp_pktout_queue_param_t out_queue_param;
39   odp_pktio_t pktio;
40   odp_pktio_config_t config;
41
42   odp_pktio_param_init(&pktio_param);
43
44   pktio = odp_pktio_open(name, pool, &pktio_param);
45   if (pktio == ODP_PKTIO_INVALID) {
46     printf("Failed to open %s\n", name);
47     exit(1);
48   }
49
50   odp_pktio_config_init(&config);
51   config.parser.layer = ODP_PROTO_LAYER_L2;
52   odp_pktio_config(pktio, &config);
53
54   odp_pktin_queue_param_init(&in_queue_param);
55   odp_pktout_queue_param_init(&out_queue_param);
56
57   in_queue_param.op_mode = ODP_PKTIO_OP_MT_UNSAFE;
58
59   if (odp_pktin_queue_config(pktio, &in_queue_param)) {
60     printf("Failed to config input queue for %s\n", name);
61     exit(1);
62   }
63
64   out_queue_param.op_mode = ODP_PKTIO_OP_MT_UNSAFE;
65
66   if (odp_pktout_queue_config(pktio, &out_queue_param)) {
67     printf("Failed to config output queue for %s\n", name);
68     exit(1);
69   }
70
71   if (odp_pktin_queue(pktio, pktin, 1) != 1) {
```

```
72      printf("pktin queue query failed for %s\n", name);
73      exit(1);
74    }
75    if (odp_pktout_queue(pktio, pktout, 1) != 1) {
76      printf("pktout queue query failed for %s\n", name);
77      exit(1);
78    }
79    return pktio;
80  }
81
82  static int run_worker(void *arg ODP_UNUSED)
83  {
84    odp_packet_t pkt_tbl[MAX_PKT_BURST];
85    int pkts, sent, tx_drops, i;
86    uint64_t wait_time = odp_pktin_wait_time(ODP_TIME_SEC_IN_NS);
87
88    if (odp_pktio_start(global->if0)) {
89      printf("unable to start input interface\n");
90      exit(1);
91    }
92    printf("started input interface\n");
93    if (odp_pktio_start(global->if1)) {
94      printf("unable to start output interface\n");
95      exit(1);
96    }
97    printf("started output interface\n");
98    printf("started all\n");
99
100   while (!global->exit_thr) {
101     pkts = odp_pktin_recv_tmo(global->if0in, pkt_tbl, MAX_PKT_BURST,
102             wait_time);
103
104     if (odp_unlikely(pkts <= 0)) {
105       if (global->wait_sec > 0)
106         if (!(--global->wait_sec))
107           break;
108       continue;
109     }
110
111     for (i = 0; i < pkts; i++) {
112       odp_packet_t pkt = pkt_tbl[i];
113       odph_ethhdr_t *eth;
114
115       if (odp_unlikely(!odp_packet_has_eth(pkt))) {
116         printf("warning: packet has no eth header\n");
117         return 0;
118       }
119       eth = (odph_ethhdr_t *)odp_packet_l2_ptr(pkt, NULL);
120       eth->src = global->src;
121       eth->dst = global->dst;
122     }
123     sent = odp_pktout_send(global->if1out, pkt_tbl, pkts);
124     if (sent < 0)
125       sent = 0;
126     tx_drops = pkts - sent;
127     if (odp_unlikely(tx_drops))
128       odp_packet_free_multi(&pkt_tbl[sent], tx_drops);
129   }
130
131   return 0;
132 }
133
134 int main(int argc, char **argv)
135 {
136   odph_helper_options_t helper_options;
137   odp_pool_t pool;
138   odp_pool_param_t params;
139   odp_cpumask_t cpumask;
140   odph_odpthread_t thd[MAX_WORKERS];
141   odp_instance_t instance;
142   odp_init_t init_param;
143   odph_odpthread_params_t thr_params;
144   odph_ethaddr_t correct_src;
145   uint32_t mtu1, mtu2;
146   odp_shm_t shm;
147
148   /* Let helper collect its own arguments (e.g. --odph_proc) */
149   argc = odph_parse_options(argc, argv);
150   if (odph_options(&helper_options)) {
151     printf("Error: reading ODP helper options failed.\n");
152     exit(EXIT_FAILURE);
153   }
154
155   odp_init_param_init(&init_param);
156   init_param.mem_model = helper_options.mem_model;
157
158   if (odp_init_global(&instance, &init_param, NULL)) {
159     printf("Error: ODP global init failed.\n");
160     exit(1);
161   }
```

```
162
163   if (odp_init_local(instance, ODP_THREAD_CONTROL)) {
164     printf("Error: ODP local init failed.\n");
165     exit(1);
166   }
167
168   /* Reserve memory for args from shared mem */
169   shm = odp_shm_reserve("_appl_global_data", sizeof(global_data_t),
170             ODP_CACHE_LINE_SIZE, 0);
171   global = odp_shm_addr(shm);
172   if (global == NULL) {
173     printf("Error: shared mem alloc failed.\n");
174     exit(EXIT_FAILURE);
175   }
176
177   memset(global, 0, sizeof(global_data_t));
178   global->shm = shm;
179
180   if (argc > 7 ||
181       odph_eth_addr_parse(&global->dst, argv[3]) != 0 ||
182       odph_eth_addr_parse(&global->src, argv[4]) != 0) {
183     printf("Usage: odp_l2fwd_simple eth0 eth1 01:02:03:04:05:06"
184            " 07:08:09:0a:0b:0c [-t sec]\n");
185     printf("Where eth0 and eth1 are the used interfaces"
186            " (must have 2 of them)\n");
187     printf("And the hexadecimal numbers are destination MAC address"
188            " and source MAC address\n");
189     exit(1);
190   }
191   if (argc == 7 && !strncmp(argv[5], "-t", 2))
192     global->wait_sec = atoi(argv[6]);
193
194   if (global->wait_sec)
195     printf("running test for %d sec\n", global->wait_sec);
196
197   /* Create packet pool */
198   odp_pool_param_init(&params);
199   params.pkt.seg_len = POOL_SEG_LEN;
200   params.pkt.len     = POOL_SEG_LEN;
201   params.pkt.num     = POOL_NUM_PKT;
202   params.type        = ODP_POOL_PACKET;
203
204   pool = odp_pool_create("packet pool", &params);
205
206   if (pool == ODP_POOL_INVALID) {
207     printf("Error: packet pool create failed.\n");
208     exit(1);
209   }
210
211   global->if0 = create_pktio(argv[1], pool, &global->if0in,
212            &global->if0out);
213   global->if1 = create_pktio(argv[2], pool, &global->if1in,
214            &global->if1out);
215
216   /* Do some operations to increase code coverage in tests */
217   if (odp_pktio_mac_addr(global->if0, &correct_src, sizeof(correct_src))
218       != sizeof(correct_src))
219     printf("Warning: can't get MAC address\n");
220   else if (memcmp(&correct_src, &global->src, sizeof(correct_src)) != 0)
221     printf("Warning: src MAC invalid\n");
222
223   odp_pktio_promisc_mode_set(global->if0, true);
224   odp_pktio_promisc_mode_set(global->if1, true);
225   (void)odp_pktio_promisc_mode(global->if0);
226   (void)odp_pktio_promisc_mode(global->if1);
227
228   mtu1 = odp_pktin_maxlen(global->if0);
229   mtu2 = odp_pktout_maxlen(global->if1);
230   if (mtu1 && mtu2 && mtu1 > mtu2)
231     printf("Warning: input MTU bigger than output MTU\n");
232
233   odp_cpumask_default_worker(&cpumask, MAX_WORKERS);
234
235   memset(&thr_params, 0, sizeof(thr_params));
236   thr_params.start    = run_worker;
237   thr_params.arg      = NULL;
238   thr_params.thr_type = ODP_THREAD_WORKER;
239   thr_params.instance = instance;
240
241   signal(SIGINT, sig_handler);
242
243   odph_odpthreads_create(thd, &cpumask, &thr_params);
244   odph_odpthreads_join(thd);
245
246   if (odp_pktio_stop(global->if0) || odp_pktio_close(global->if0)) {
247     printf("Error: failed to close interface %s\n", argv[1]);
248     exit(EXIT_FAILURE);
249   }
250   if (odp_pktio_stop(global->if1) || odp_pktio_close(global->if1)) {
251     printf("Error: failed to close interface %s\n", argv[2]);
```

```
252     exit(EXIT_FAILURE);
253   }
254
255   if (odp_pool_destroy(pool)) {
256     printf("Error: pool destroy\n");
257     exit(EXIT_FAILURE);
258   }
259
260   if (odp_shm_free(global->shm)) {
261     printf("Error: shm free global data\n");
262     exit(EXIT_FAILURE);
263   }
264
265   if (odp_term_local()) {
266     printf("Error: term local\n");
267     exit(EXIT_FAILURE);
268   }
269
270   if (odp_term_global(instance)) {
271     printf("Error: term global\n");
272     exit(EXIT_FAILURE);
273   }
274
275   return 0;
276 }
```