UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Fabricio Eduardo Rodríguez Cesén

Design, Implementation and Evaluation of IPv4/IPv6 Longest Prefix Match support in Multi-Architecture Programmable Dataplanes

Projeto, Implementação e Avaliação do Suporte de Casamento com Prefixo Mais Longo para IPv4/IPv6 em Planos de Dados Programáveis Multi-Arquitetura

Campinas

2018

Fabricio Eduardo Rodríguez Cesén

Design, Implementation and Evaluation of IPv4/IPv6 Longest Prefix Match support in Multi-Architecture Programmable Dataplanes

Projeto, Implementação e Avaliação do Suporte de Casamento com Prefixo Mais Longo para IPv4/IPv6 em Planos de Dados Programáveis Multi-Arquitetura

Dissertation presented to the Faculty of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Master, 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 exigidos para a obtenção do título de Mestre em Engenharia Elétrica, na Área de Engenharia de Computação.

Supervisor: Prof. Dr. Christian Rodolfo Esteve Rothenberg

Este exemplar corresponde à versão final da tese defendida pelo aluno Fabricio Eduardo Rodríguez Cesén, e orientada pelo Prof. Dr. Christian Rodolfo Esteve Rothenberg

Campinas

2018

Agência(s) de fomento e nº(s) de processo(s): Não se aplica.
ORCID: https://orcid.org/0000-0003-1165-2808

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca da Área de Engenharia e Arquitetura
Elizangela Aparecida dos Santos Souza - CRB 8/8098

R618d

Rodríguez Cesén, Fabricio Eduardo, 1989-
Design, implementation and evaluation of IPv4/IPv6 longest prefix match support in multi-architecture programmable dataplane / Fabricio Eduardo Rodríguez Cesén. – Campinas, SP : [s.n.], 2018.

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

1. Redes definidas por software (Tecnologia de rede de computador). 2. Software - Desempenho. 3. Redes locais de computação - Avaliação. 4. Roteamento (Administração de redes de computadores). 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, implementação e avaliação do suporte de casamento com prefixo mais longo para IPv4/IPv6 em planos de dados programáveis multi-arquitetura
Palavras-chave em inglês:
Software defined networking
Network performance
Computer network performance evaluation
Table lookup
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Christian Rodolfo Esteve Rothenberg [Orientador]
Marco Aurélio Amaral Henriques
Fábio Luciano Verdi
Data de defesa: 18-12-2018
Programa de Pós-Graduação: Engenharia Elétrica

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Fabricio Eduardo Rodríguez Cesén RA: 163682

Data da Defesa: 18 de dezembro de 2018

Título da Tese: “Design, Implementation and Evaluation of IPv4/IPv6 Longest Prefix Match support in Multi-Architecture Programmable Dataplanes

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 PósGraduação da Faculdade de Engenharia Elétrica e de Computação.

To my wife, Aline, who has been a constant source of support and encouragement.

Acknowledgements

First and foremost, I have to thank my wife for her love and support. An special thank to my family, despite the difficulties of being far away, they always have been supporting me and encouragement to continue with my dreams.

My parents, Mariana and Vinicio, who have always supported me unconditionally and whose good examples have taught me to work hard for the things that I aspire to achieve.

I would like to sincerely thank my advisor, Professor Christian, for his guidance and support throughout this study and especially for his confidence in me.

To all my friend and colleagues, they have been an important part of this process, with their advice and support.

This work was supported by the Innovation Center, Ericsson Telecomunicações S.A., Brazil under grant agreements UNI.61 and UNI.63.

“The most exciting phrase to hear in science, the one that heralds new discoveries, is not ‘Eureka!’ but ‘That’s funny...’ ”

(Isaac Asimov)

Abstract

Among the New trends in dataplane programmability inside Software Defined Networking (SDN) stand out the efforts to bring multi-platform support with a high definition of the information that is processed by the dataplane pipeline. The Programming Protocol-Independent Packet Processors (P4) Domain Specific Language (DSL) is an emerging trend to express how the packets are processed by the dataplanes of a programmable network platform. In parallel, OpenDataPlane (ODP) project creates an open-source, cross-platform set of Application Programming Interfaces (APIs) designed for the networking dataplane. Multi-Architecture Compiler System for Abstract Dataplane (MACSAD) is an approach to converge P4 and ODP in a conventional compilation process, achieving portability of the dataplane applications without affecting the target performance improvements.

This thesis aims at adding IPv4/IPv6 Longest Prefix Match (LPM) support to MACSAD integrated with ODP APIs and P4 programmability delivering high-performance dataplane capabilities. The proposed LPM support for MACSAD combines the lookup algorithm and the ODP API library with MACSAD table support, to create a complete forwarding base used in the LPM process. The IPv4 implementation adapts the current ODP lookup algorithm to work with MACSAD. IPv6 lookup implementation, currently not supported by ODP, it is an extension of the IPv4 support developed using the same algorithm adapted to a 128-bit key. For the performance evaluation of the LPM support, we use a traffic generator tool Network Function Performance Analyzer (NFPA) that allows generating different types of traffic across MACSAD. Experimental results show that it is possible to reach a throughput of 10G with packets sizes of 512 Bytes and above. As a side contribution on this front is the open source BB-Gen packet crafter tool.

Keywords: P4; Software Defined Networking; Performance analysis; Programmable networks.

Resumo

Dentre as novas tendências em programação de dataplane dentro de SDN (Software Defined Networking) destacam-se os esforços para prover um suporte multi-plataforma dotado de alta definição das informações que são processadas pelo pipeline do plano de dados. Como forma de mitigar tais problemas, verifica-se que a Linguagem Específica de Domínio (DSL) Programming Protocol-Independent Packet Processors (P4) desponta como uma tendência emergente para expressar como os pacotes são processados pelo plano de dados de uma plataforma de rede programável. De modo independente e em paralelo, constata-se que o projeto OpenDataPlane (ODP) cria um conjunto de plataformas abertas de Application Programming Interfaces (APIs) projetado para o plano de dados de rede. Isso posto, tem-se que o Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD) surge como uma abordagem para convergir P4 e ODP em um processo de compilação convencional, arquivando a portabilidade dos aplicativos de plano de dados sem afetar as melhorias de desempenho do alvo.

Este trabalho tem como objetivo adicionar o suporte do Longest Prefix Match (LPM) do IPv4/IPv6 ao MACSAD, integrado com as APIs do ODP e à programação P4, oferecendo recursos de planejamento de dados de alto desempenho. O suporte ao LPM proposto para o MACSAD combina o algoritmo de lookup e a biblioteca da API do ODP com o suporte à tabela MACSAD, para criar uma base de encaminhamento completa usada no processo do LPM. A implementação do IPv4 adapta o atual algoritmo de lookup do ODP para trabalhar com o MACSAD. A implementação de lookup IPv6, atualmente não suportada pelo ODP, é uma extensão do suporte IPv4 que é desenvolvido usando o mesmo algoritmo adaptado a uma chave de 128 bits. Para a avaliação de desempenho do suporte ao LPM, utilizamos uma ferramenta geradora de tráfego Network Function Performance Analyzer (NFPA) que permite gerar diferentes tipos de tráfego no MACSAD. Cabe ainda destacar, como uma contribuição lateral deste trabalho, o desenvolvimento da ferramenta geradora de pacote BB-Gen, já com lançamento open source. Resultados experimentais mostram que é possível atingir um throughput de 10G com tamanhos de pacotes de 512 bytes ou superiores.

Palavras-chaves: P4; Rede Definida por Software; Desempenho; Redes Programáveis.

List of Figures

List of Tables

Acronyms

API Application Programming Interface.

ARM Advanced RISC Machine.

BPF Berkeley Packet Filter.

bps bits per second.

DApp Dataplane Application.

DPDK Data Plane Development Kit.

DSL Domain Specific Language.

DUT Device Under Test.

eBPF Extended BPF.

FIB Forwarding Information Base.

FPGA Field Programmable Gate Array.

GCC GNU Compiler Collection.

GRE Generic Routing Encapsulation.

HAL Hardware Abstraction Library.

HLIR High Level IR.

IR Intermediate Representation.

LLVM Low Level Virtual Machine.

LPM Longest Prefix Match.

MacS MACSAD Switch.

MACSAD Multi-Architecture Compiler System for Abstract Dataplanes.

MPLS Multiprotocol Label Switching.

NFPA Network Function Performance Analyzer.

ODP OpenDataPlane.

OF OpenFlow.

OVS OpenvSwitch.

P4 Programming Protocol-Independent Packet Processors.

PI Protocol Independence.

PISA Protocol Independent Switch Architecture.

pps packets per second.

PRT P4Runtime.

SAI Switch Abstraction Interface.

SDK Software Development Kit.

SDN Software Defined Networking.

SoC System on a Chip.

T4P4S Translator for P4 Switches.

TCAM Ternary Content Addressable Memory.

TTL Time to Live.

VXLAN Virtual eXtensible Local Area Network.

Contents

1 Introduction

Hundreds of thousands of packets are being transmitted through the Internet every second. As a consequence, the requirements on the network are increasing exponentially, and it is necessary to evaluate the actual capabilities and how to manage this traffic efficiently. A flexible, re-designable and configurable network is needed to ensure the possibility to change the features of the dataplane.

Software Defined Networking (SDN) (KREUTZ *et al.*, 2014) is an emerging network architecture where the network splits into control and forwarding plane. This migration of control, which is tightly bounded in particular network devices, is converted into available computing devices, enabling the underlying abstracted infrastructure for applications and network services capable to treat the network as a logical or virtual entity. The first communications standard interface defined between the control and forwarding layers was OpenFlow (OF). It allows direct access to manipulation of the forwarding layer of physical and virtual network devices such as routers and switches. However, OpenFlow has some limitations, for example, it needs to know the types of the packet headers introducing difficulties in implementing new protocols and headers. Therefore, to enable programming the forwarding chip to support new protocols, a new protocol is proposed with independent programming abstractions, such Programming Protocol-Independent Packet Processors (P4) (BOSSHART et al., 2014).

P4 is an open source language for expressing how the packets are processed by the pipeline of a network forwarding element. It is based on a Match+Action forwarding model, and it works together with SDN protocols. Use of P4 makes it possible to reconfigure the packet's fields, and the programmers can change the way these ones flow on the switch. What is more, with target independence, the programmers do not have to know the specifications of the underlying hardware for being able to describe packet processing functionality. With the use of protocol independent, the switches do not need to be bound to any specific network protocols (BOSSHART et al., 2014). With SDN, new protocols program the routing devices that have emerged, as well as new data plan hardware and languages, such P4. On the other hand, ODP (OPENDATAPLANE, 2013c) provides an open-source, cross-platform set of Application Programming Interfaces (APIs) for the networking dataplane. The APIs enable developers to create dataplane codes across different targets, being neutral to different vendors and platforms.

With P4 and ODP working together, it is possible to determine and program dataplanes beyond multiple targets with a common compiler system. MACSAD (PATRA *et al.*, 2016; PATRA *et al.*, 2017) is an approach to accommodate P4 and ODP through a common compilation process delivering portability of dataplane applications without compromising target performance improvements, translating P4-defined dataplanes into high-level ODP APIs. MACSAD has a designed compiler module that generates an Intermediate Representation (IR) for P4 applications. However, there is no previously published study in where any user communities or third-parties presented ODP IP Lookup support for IPv6, and MACSAD has limited support for IPv4 forwarding. In our work, we design, implement and evaluate the IPv4/IPv6 Longest Prefix Match (LPM) support for a multi-architecture compiler system (MACSAD) capable of supporting new Domain Specific Languages (DSLs) and network platforms. The performance of the prototype is tested in different dataplane instances for a 10G setup and now it is implemented using on ODP Layer 3 forwarding base, IPv4 LPM support for MACSAD. Using the mentioned base, we adapt the table management from ODP and combine with our forwarding support. Some functions considered to integrate the lookup tables from MACSAD to the ODP and with the LPM algorithm. Notable works regarding IP Lookup implementations are present in (RéTVáRI *et al.*, 2013; SHAHHAZ et al., 2016; PONG; TZENG, 2012).

The generation of packet traces is commonly required to carry out performance evaluations and hence it becomes a relevant task for the development of new network solutions and the evaluation of the existing ones. Our analysis found a gap among open source PCAP trace generators with simple interfaces and covering the relevant requirements for rich performance experiments. Some alternatives support a complete set of protocols and allow to generate different sets of PCAP traces but do not provide important traffic flow characteristics (e.g., packet size and address distribution). Other tools support different encapsulation protocols but not Virtual eXtensible Local Area Network (VXLAN) and Generic Routing Encapsulation (GRE) together in the same tool. Although there are many tools for packet generation, the complementary creation of table traces remains limited. Table traces are necessary in a programmable network to fill the table flow configuration of the network devices.

Our need for a single packet generator tool to meet the packet trace requirements of our research use cases to evaluate the performance and scalability of programmable dataplanes (PATRA *et al.*, 2016; PATRA *et al.*, 2017; PATRA *et al.*, 2018) and identify the limitations of existing tools, motivated our work to develop a packet generator prioritizing two essential characteristics: (*i*) simple to use, (*ii*) wide protocol support and rich customizability. To that end, as our baseline, we opted for the well known Scapy python library (BIONDI, 2008) for easier extension and packet manipulation.

1.1 Research Goals

The following section describes the current limitations with MACSAD IP lookup process along with the research objectives.

1.1.1 Problem Definition

The MACSAD implementation allows performing a basic Layer 2 forwarding. However, it limits the use-cases with the switch and the capabilities to perform a Layer 3 packet forwarding. The current limitations of IP lookup support in MACSAD are the following ones:

1. **Limited support of IPv4 lookup**. There is no IPv4 lookup algorithm implementation featuring the ODP APIs, resulting MACSAD to support only Layer 2 routing. Therefore, the implementation of an IP lookup LPM support expanding the use-cases for the project has become a necessity.

2. **No IPv6 support in ODP**. ODP helper library provides support for IPv4 lookup forwarding. However, there is not an actual implementation of IPv6 lookup provided by ODP. No short-term plans related to this implementation by the ODP team, hence, a high-reliability multi-protocol (IPv4/IPv6) lookup mechanism is imperative to perform the lookup process.

3. **Performance evaluation**. An evaluation of the Switch (MACSAD) with complete IPv4/IPv6 lookup process is indispensable. It has to maintain the complete capabilities of the network. Without affecting the throughput of the network independently of the LPM process, it has to conserve a high-reliability forwarding.

1.1.2 Objectives

To address the identified issues, the main objective of this work is to design, implement and evaluate the Longest Prefix Match (LPM) IPv4/IPv6 support in MACSAD. To this end, the following specific objectives are identified:

- **IPv4 LPM support for MACSAD.** Adapting the IP lookup algorithm to interact with ODP APIs. To provide a mechanism integrating an algorithm for IP lookup and ODP APIs in conjunction with MACSAD to perform the LPM task. Maintaining MACSAD architecture base and using ODP APIs allow us to create a multi-target platform.

- **IPv6 LPM support for MACSAD.** To extend the support of IPv4 lookup and make adapting and integrating the IPv6 LPM process plausible. Creating a complete ODP library that supports IPv6 lookup, contributing in this way with the ODP project.

- **Controller support.** To complete the lookup process and following the SDN design, the controller has to interact with MACSAD to manage the packet traffic and table actions management.
- **Workload generation.** To have a broad and deep experimental evaluation, we need to reach packet traces with different characteristics such as specific header information, number of entries and, packet size.
- **Performance evaluation.** With the lookup process, the performance of the network cannot be strongly affected. We envision that implementing the correct algorithm, the performance of MACSAD will become optimal, exploiting the full capabilities of the network. Evaluating with unidirectional traffic and the packet configurations traces, with fixed destination and source MAC addresses, IP addresses and ports.

1.2 Methodology and Contributions

To achieve the objectives, the following main activities were planned: (i) Literature review, (ii) Binary tree and ODP LPM library implementation, (iii) MACSAD lookup implementation, and (iv) performance evaluation.

- **Literature review.** It compels the study and analysis of the state of the art of IPv4/IPv6 LPM algorithms to support a Multi-Architecture Programmable Dataplane. The study of ODP and P4 language to be used for the LPM implementation. The study of the architecture of MACSAD and the current project's support.
- **Binary tree and ODP implementation.** After the literature review, we implement the Binary tree and the ODP API library. As a first part of the process, the IPv4 support is developed, a second part consist of the IPv6 implementation, extending the lookup support and capabilities.
- **Lookup implementation.** With the binary tree and ODP library implemented, the next step planned consist to develop the LPM support for MACSAD and the integration with the controller. The implementation separates the IPv4 and IPv6 support as two processes being developed before the corresponding binary tree and ODP implementation.
- **Performance evaluation.** Using the Network Function Performance Analyzer (NFPA) tool, we evaluate MACSAD for different IPv4/IPv6 traces and platform configurations (e.g., CPU cores, I/O).

The contributions of this dissertation can be summarized as follows:

- Implementation and experimental evaluations of IPv4 and IPv6 Longest Prefix Match (LPM) support in MACSAD.
- Prototype implementation of a new ODP Helper library for the IPv6 lookup mechanism based on the current IPv4 solution and evaluation of the performance and scalability for diverse workloads and target platform configurations.
- The performance and scalability evaluation of MACSAD pipelines analyzing the impact of varying packet sizes and memory lookup tables, and investigating the impact of increased pipeline complexity of $P4_{14}$ and $P4_{16}$ use cases ranging from Ethernet and IPv4/v6 to VXLAN-based Data Center Gateway and Broadband Network Gateway (BNG). Combining the high-level Protocol Independence (PI) programmability of P4 with the low-level cross-platform (HW & SW) APIs brought by ODP.
- A P4 CLI-based packet crafter to generate packet flows formatted as PCAP files. Supporting different standard protocols and able to create the necessary traces for network function configuration and testing from a P4 file as an Input.
- Open source artifacts. The MACSAD code including IPv4 and IPv6 use cases were open sourced. BB-Gen tool, the PCAPs, and Traces generated are open for developers to evaluate the performance of the projects, being simple to integrate for different trace formats.

Contributions of this thesis have led to seven co-authored publications (see Appendix A).

1.3 Text Organization

The rest of this work contains the following topics. Background information including related work as presented in Chapter 2. Chapter 3 describes our architecture proposal to design and implement of IPv4/IPv6 LPM support for MACSAD, with the details of each process involved in its development. Chapter 4 presents the packet generator tool that will be used for the performance evaluation. Chapter 5 shows our performance evaluation in different scenarios. Finally, our conclusions and future work are presented.

2 Literature Review

In this chapter, we review relevant literature for our research and related works about problems and solutions that carries close resemblance to our approach.

2.1 Background

This section defines four main concepts in our research work: P4 expressing how the pipeline processes packets, ODP as the APIs for the networking dataplane, NFPA as open-source measurement application and finally a MACSAD.

2.1.1 Programming Protocol-Independent Packet Processors

Protocol Independent Switch Architecture (PISA) (MCKEOWN, 2016; GUREVICH, 2015) allows a custom definition of network protocols in a switch design approach. Top-down these devices allow us to tell them how to process packets (GUREVICH, 2015). The forwarding plane does not know any protocols until it is programmed. In Figure 1 is shown the logical mapping dataplane design for physical resources. One of the principal characteristics of P4 allows to reconfigure PISA devices in the field (and eventually on the fly) (MCKEOWN, 2016).

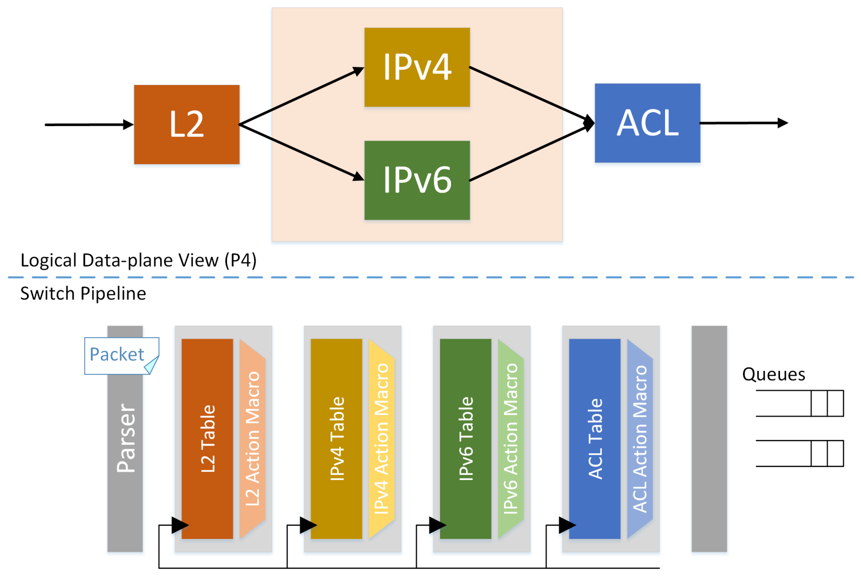


Figure 1 – Mapping to Physical Resources. Source (GUREVICH, 2015)

Figure 2 shows the capability of P4 to map custom protocols together with defined protocols to the physical resources.

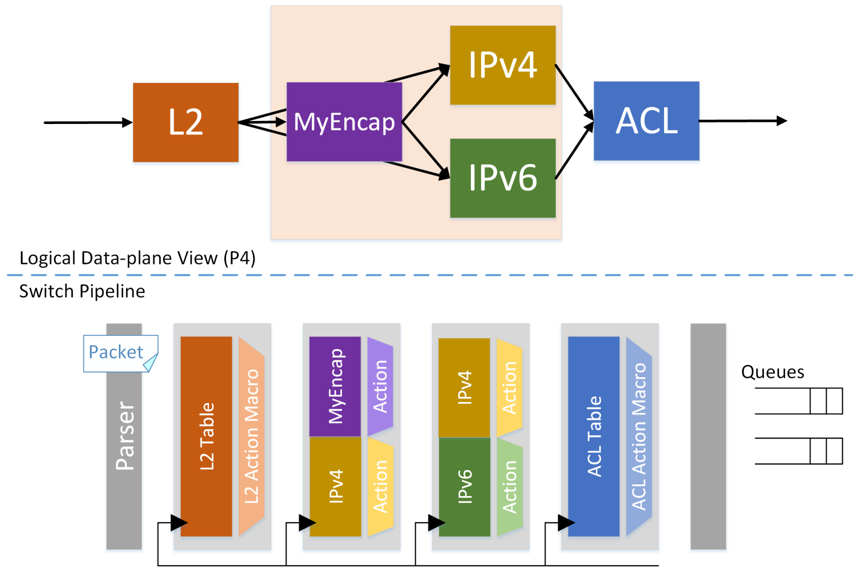


Figure 2 – Mapping custom protocol to Physical Resources. Source (GUREVICH, 2015)

P4 is an innovation providing an abstract model suitable for programming the network dataplane (BOSSHART et al., 2014). A P4-enabled device is protocol independent. It delineates the packet headers and specifies the packet parsing and processing behaviors.

A P4 program include the flowing elements:

1. **Header definitions,** that specify the field names and widths for protocol headers (see listing 2.1).

```
header_type ethernet_t {
  fields {
      dstAddr : 48;      // Destination MAC address
      srcAddr : 48;      // Source MAC address
      etherType : 16;      // Ethernet type
  }
}
header ethernet_t ethernet;
```

Listing 2.1 – P4 header definition example

2. **Metadata,** holds information about the packet that is not normally represented by the packet data (see listing 2.2).

```
header_type local_metadata_t {
  fields {
    cpu_code : 8;        // Code for packet going to CPU
    port_type : 2;       // Type of port
    ingress_error : 1;     // An error in ingress port check
    was_mtagged : 1;     // Track if pkt was mtagged on ingr
  }
}
```

```
metadata local_metadata_t local_metadata;
```

Listing 2.2 – P4 metadata definition example

3. **Registers, Meters, and Counters,** for state independent of packets (see listing 2.3 and 2.4).

```
counter ip_pkts_by_dest {
  type : packets;
  direct : ip_host_table;
}
```

Listing 2.3 – P4 counter definition example

```
meter customer_meters {
  type : bytes;
  instance_count : 1000;
}
```

Listing 2.4 – P4 meter definition example

4. **Packet parser specification,** that generates a Parsed Representation where the match+action tables operate (see listing 2.5).

```
parser start {
  return parse_ethernet;
}

parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
  0x0800 : parse_ipv4;
  default: ingress;  }
}

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

Listing 2.5 – P4 parser example

5. **Match-action table specification,** identifying the packet and metadata fields to be read and the possible actions to execute in response (see listing 2.6).

```
table sendout {
  reads {
    standard_metadata.egress_port : exact;
  }
  actions {
    on_miss;
    rewrite_src_mac;
  }
  size : 512;
}
```

Listing 2.6 – P4 Match-action table specification example

6. **Actions,** functions that may be parameterized and that invoke one or more primitives (see listing 2.7).

```
action on_miss() {
}

action rewrite_src_mac(smac) {
  modify_field(ethernet.srcAddr, smac);
}
```

Listing 2.7 – P4 actions specification example

7. **Control flow,** indicating the table execution sequence, with support for conditional branching (see listing 2.8).

```
control ingress {
  apply(sendout);
}
```

Listing 2.8 – P4 control flow specification example

Figure 3 represents the abstract forwarding model, illustrating how the pipeline of a network forwarding element should process packets (SHAHHAZ et al., 2016). When a packet is received, the headers enter into the parser pipeline then they pass through the match+action tables flow, and finally the headers return back, and the packet is sent.

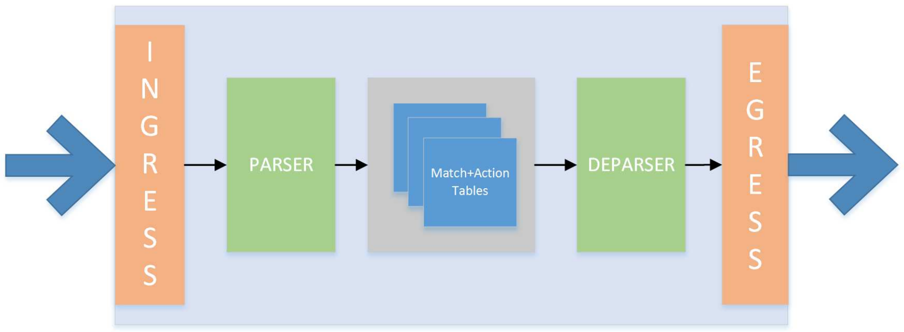


Figure 3 – P4 Abstract Forwarding Model. Source: Adapted from (MCKEOWN, 2016)

In the P4 program, all these activities are defined inside the ingress and egress of control flows. This is possible due to the protocol independent feature that details custom headers expressing arbitrary network protocol headers and fields.

2.1.2 OpenDataPlane

ODP project is a networking dataplane API specification. It allows application developers to write and implement dataplane applications. It can leverage portability and multi-platform support, besides the use of specific hardware acceleration capabilities. ODP defines a set of high-level common APIs (e.g., CPU control, operations on a packet Input/Output interface, operations on memory), bringing the opportunity of span standard features across multi-targets (e.g. Advanced RISC Machine (ARM)v7, ARMv8, MIPS64, Power, and x86) making dataplane applications portable. Table 1 presents the ODP supported platforms.

The principal attributes of the ODP API are summarized below:

- Open Source and open for contributions.
- Vendor and platform independent.
- Application-centric, encloses the functional needs of dapaplane applications.
- Allows portability.
- Defined in conjunction with application writers and platform implementers.
- Architecture efficiently used on a wide range of different platforms.
- Sponsored, governed, and maintained by the Linaro Networking Group (LNG).

Figure 4 presents the ODP software stack. ODP is composed by a common layer (ODP APP) and the implementation layer (ODP Implementation). The Applications defined in the common layer are portable across all ODP implementations. An application to be executed by the ODP API will be linked to the ODP implementation for the specific execution platform. The purpose of the implementation layer is to provide a mapping of ODP APIs to the underlying capabilities (i.e., hardware co-processing and acceleration

Table 1 – ODP supported platforms

Company	**Supported Platforms**
Cavium	ThunderX CN88xx 24-48core ARMv8 OCTEON TX CN83/81xx 1-24 core ARMv8
Kalray	MPPA
Freescale	QorIQ – ARM based DPAA2 architecture LS2080, LS2085 QorIQ – ARM & PowerPC based DPAA architecture LS1043
Texas Instrument	Keystone2 Cortex A15
Linaro	PCIe NIC optimized implementation (odp-dpdk)
Marvell	Marvell ARMADA SoC Implementation

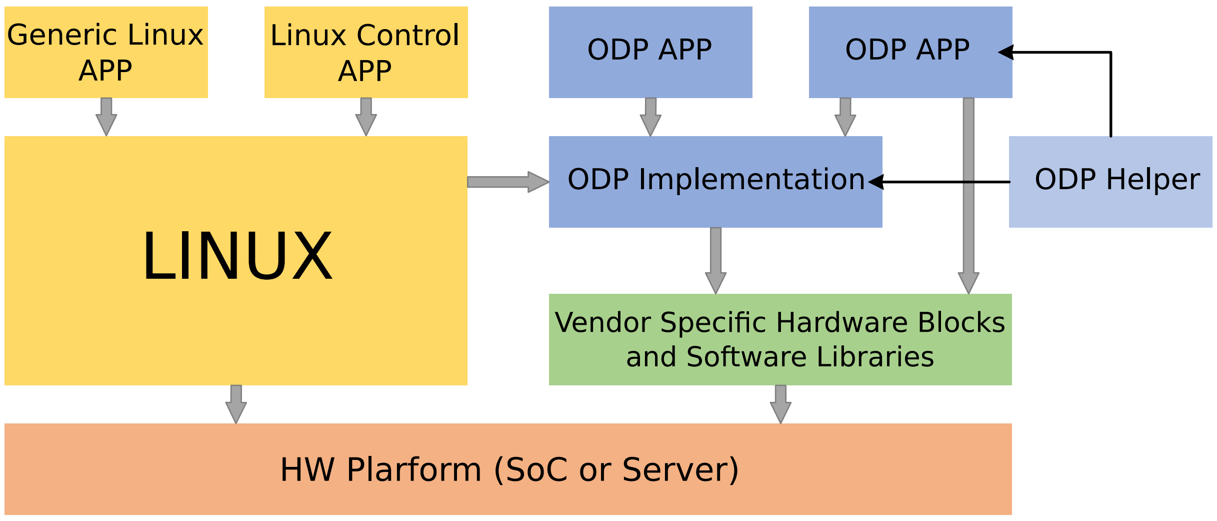

Generic Linux APP
Linux Control APP
ODP APP
ODP APP
LINUX
ODP Implementation
ODP Helper
Vendor Specific Hardware Blocks and Software Libraries
HW Plarform (SoC or Server)

Figure 4 – ODP software stack in a Linux-based target. Source (OPENDATAPLANE, 2013c)

support) of System on a Chips (SoCs) hosting ODP implementations. The implementation has been optimized for a particular hardware platform (SoC or Server). It will call the Vendor Specific Hardware Blocks and Software Libraries (Software Development Kit (SDK)) through an inline call and takes advantage of the vendor extension that it is not yet standardized (OPENDATAPLANE, 2013c).

ODP consists of a common layer and an implementation layer. Applications written to the common layer are portable across all ODP implementations. To compile and run an ODP application, it is compiled against a specific ODP implementation layer.

The purpose of the implementation layer is to provide an optimal mapping of ODP APIs to the underlying capabilities (including hardware co-processing and acceleration support) of SoCs hosting ODP implementations. As a bootstrapping mechanism for applications, as well as to provide a model for ODP implementers, ODP provides a 'linux-generic' reference implementation designed to run on any SoC which has a Linux kernel. While linux-generic is not a performance target, it does provide a starting point for ODP implementers and application programmers alike. As a pure software implementation of ODP, linux-generic is designed to provide best-in-class performance for general Linux data plane support.

ODP helper library (OPENDATAPLANE, 2013a) offers a basic support for table management (lookup mechanism), contrary to Data Plane Development Kit (DPDK), which has various fully optimized table management libraries. It also offers support to cuckoo, hash, IP lookup and linear tables.

ODP has been taken up by vendors that provide much more functionalities in their hardware than the plain NIC capabilities. For instance the ODP-DPDK (OPENDATAPLANE, 2013b) packet, as a NIC + ODP library, helping the migration from the lower

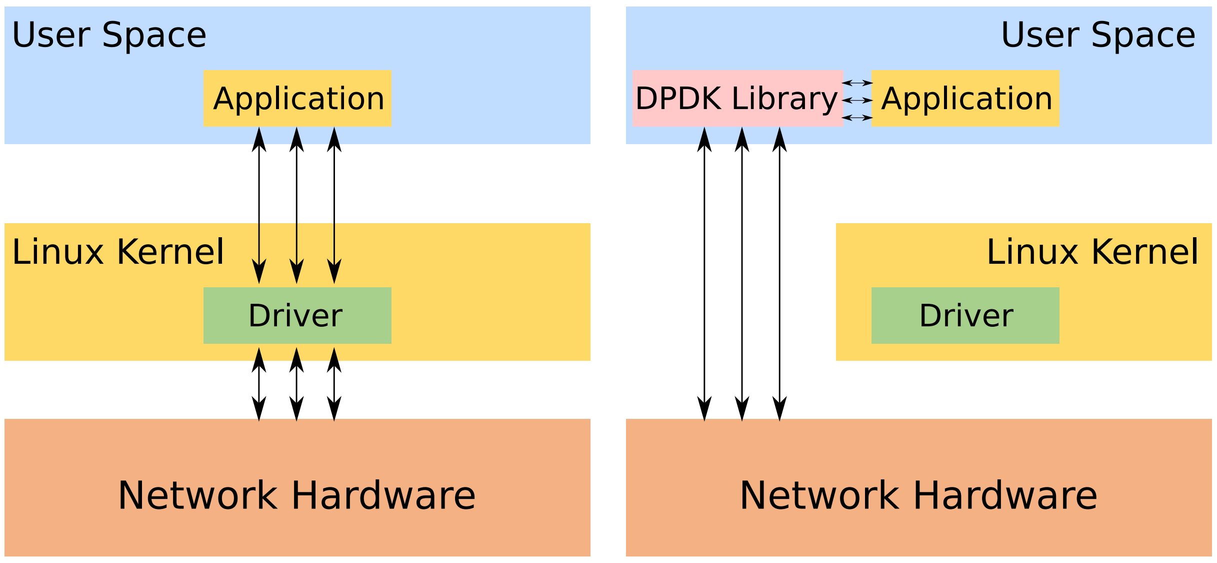


Figure 5 – Linux Kernel without DPDK vs Linux Kernel with DPDK

level of DPDK API to the ODP abstraction.

To improve the performance, ODP can use the user-space fast packet processing I/O support for traditional NICs. Odp-linux (a functional reference targeting simplicity over performance) has PKTIO support for Netmap (RIZZO, 2012) and DPDK (INTEL, 2014). With different characteristics and architecture, the DPDK and Netmap drivers are described below:

- DPDK: it is a Linux Foundation project consisting of libraries and drivers for developers to create fast packet processing Dataplane Application (DApp). It started with x86 architecture and later expanded to other platforms like ARM and IBM POWER chips, and so forth. Figure 5 compares packet path in a vanilla linux-kernel and with DPDK driver and shows that a DApp can access network interface from user space for faster packet processing.
- Netmap: its framework allows fast packet access from the network card. It reduces packet processing cost by preallocation of resources, batch packet handling and use of shared memory to achieve higher performance. It works along with Linux kernel and allows the use of Linux tools like "ethtool", "ip" for interface management.

2.1.3 Network Function Performance Analyzer

The NFPA (CSIKOR *et al.*, 2015a; CSIKOR *et al.*, 2015b), was proposed as a benchmarking tool that allows the user to measure important performance metrics of a network function compiled on any hardware and software combination, and to compare the results collected in a public Database. All the experiments running through the NFPA tool are following standardized methodologies (BRADNER; MCQUAID, 1999).

NFPA was implemented using Python. A configuration file can configure the measurements parameters and traffic traces to be used. NFPA is built on Intel's DPDK (INTEL, 2014) to avoid the limitation of kernel space with network card drivers; for sending and receiving traffic, NFPA uses PktGen (WILES, 2010; TURULL *et al.*, 2016) with custom Lua scripts for parameterizing, automating and controlling the measurements. The results are saved in a local SQLite database and have generated Gnuplot (GNUPLOT, 1986) graphs from the performance results.

NFPA provides a complete selection of synthetic traffic traces with several packet headers and sizes to obtain realistic scenarios as possible. There are also supported custom user tracers in Pcap files. The traffic traces available by the NFPA project (CSIKOR *et al.*, 2015a) includes Layer 2, Layer 3 (IPv4), VxLAN (MAHALINGAM *et al.*, 2014) and Multiprotocol Label Switching (MPLS) (ROSEN *et al.*, 2001).

The NFPA standalone node is connected to the Device Under Test (DUT) as it is presented in Figure 6[1]. The user can set the measurement setup (e.g., details of the hardware and software components, the number of repeated measurements and their duration) and the traffic traces to use. NFPA sends packets on port 0, they are processed and forwarded in the DUT node and are received on port 1. Then the throughput of the DUT is calculated in terms of packets per second (PPS) and bits per second (BPS). Once the measurement is finished NFPA saves the performance results in the local database.

When is evaluated the performance capabilities of an Ethernet device, the principal indicator is the raw bandwidth (BPS). However, it is also important to analyze the DUT capability to switch/route as many packets as required to achieve wire rate performance. This is the PPS metric (JUNIPER NETWORKS, 2015).

To calculate the amount of PPS to archive wire-rate performance it is necessary to consider the fact that the IP protocol allows variable payload sizes. With smaller packet sizes passing over the link, it is required more packets to achieve the wire rate performance, while if the packet size increases the necessary packets to saturate the link decreases:

$$pps = \frac{Gbps}{(pkt\_size + 20) * 8}$$

During the calculation of the PPS value, we need to consider the space that each packet will occupy, for this end, we will use the frame size, as well as the Inter-frame Gap (12 Bytes), and the Preamble (8 Bytes):

$$20 = \frac{Interframe\ Gap}{12\ Bytes} + \frac{Preamble}{8\ Bytes}$$

[1] For illustration purpose MACSAD is the DUT

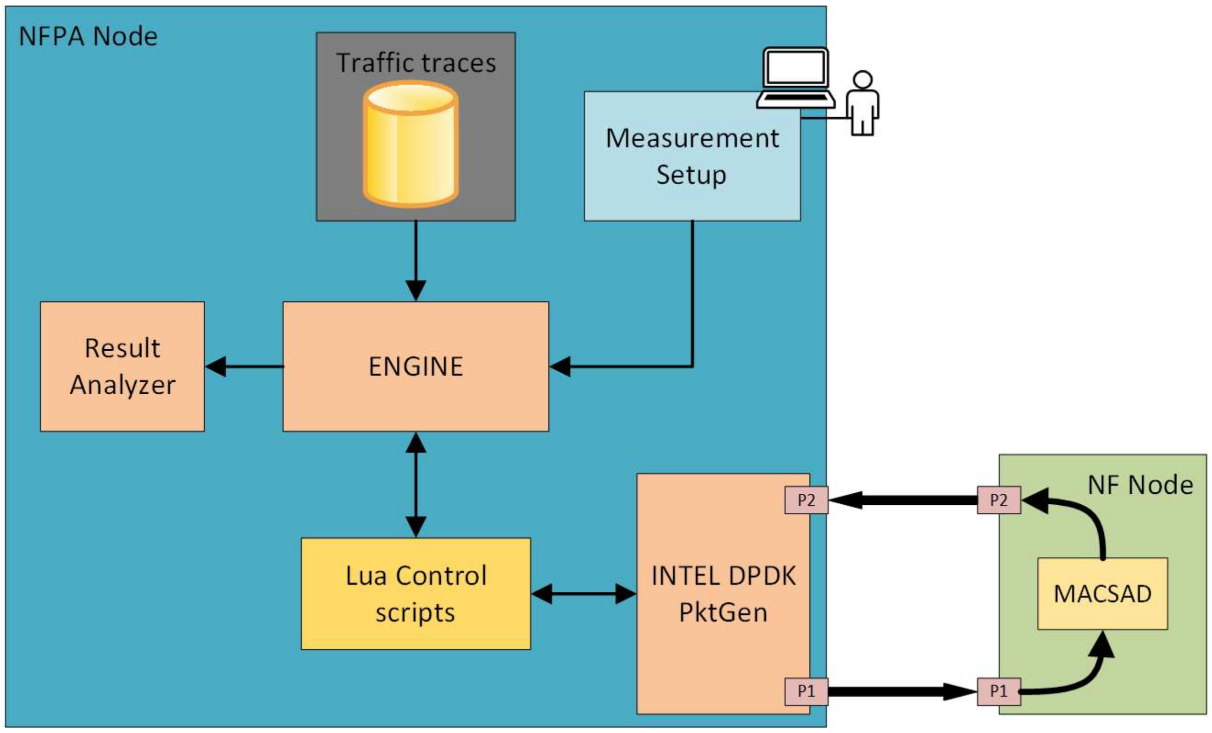


Figure 6 – NFPA Architecture. Source: Adapted from (CSIKOR *et al.*, 2015b)

If we consider the smaller packet size (64 Bytes) and a line rate throughput of 10 Gbps. It is possible to calculate the necessary PPS to saturate the link. In this case the maximum PPS is 14,880,952:

$$pps = \frac{10\ Gbps}{(64 + 20) * 8} = 14,880,952\ pps = 14.88\ Mpps$$

In table 2 are summarized the pps line rate values for different packet sizes.

Table 2 – pps Calculation for different packet sizes

Speed	**bits/second**	**bytes/second**	**Packet Size**	**Maximum PPS**
10 Gbps	10,000,000,000	1,250,000,000	64	14,880,952
10 Gbps	10,000,000,000	1,250,000,000	128	8,445,946
10 Gbps	10,000,000,000	1,250,000,000	256	4,528,986
10 Gbps	10,000,000,000	1,250,000,000	512	2,349,624
10 Gbps	10,000,000,000	1,250,000,000	1024	1,197,318
10 Gbps	10,000,000,000	1,250,000,000	1280	961,538
10 Gbps	10,000,000,000	1,250,000,000	1518	812,744

2.1.4 Multi-Architecture Compiler System for Abstract Dataplanes

MACSAD aims at taking the advantages of P4 and ODP in a common compiler system to determine and program dataplanes beyond multiple targets. MACSAD can

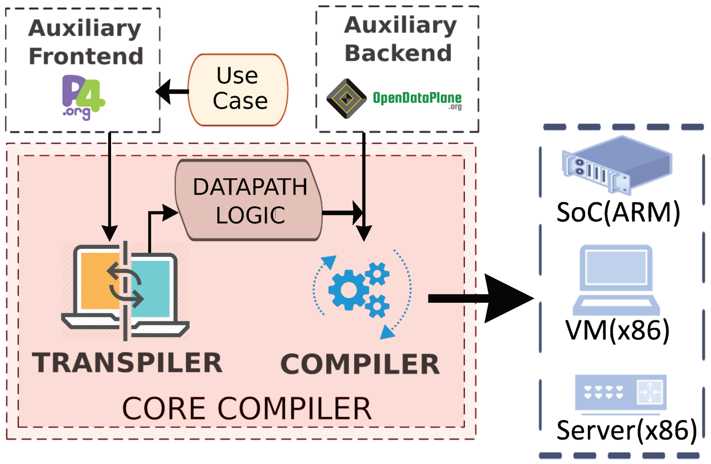


Figure 7 – MACSAD Architecture. Source (PATRA *et al.*, 2016)

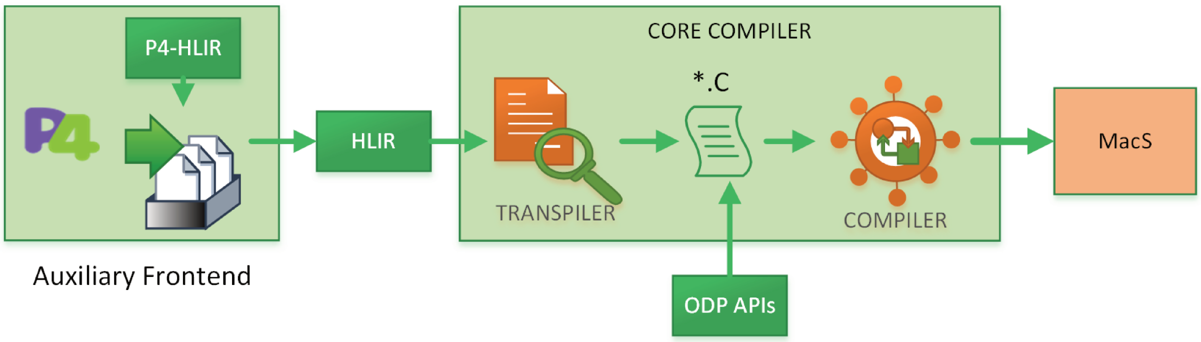


Figure 8 – Compilation Process. Source: Adapted from (PATRA *et al.*, 2016)

deliver portability of dataplane applications without compromising target performance improvements. The architecture overview is shown in Figure 7. It has three main modules: (i) Auxiliary Frontend, (ii) Auxiliary Backend and (iii) Core Compiler.

In the way that MACSAD may support different targets optimally which is a necessary solution to:

1. Support different DSLs with a plug-in framework (Auxiliary Frontend, with P4 as the premier choice).

2. Multi-platform support, linking together target-specific SDKs (Auxiliary Backend, with ODP as the desired backend).

3. Turns the intermediate representation for P4 applications generated by the auxiliary frontend, and this together with the auxiliary backend, into the imaged target (Core Compiler, compounded of a Transpiler and a Compiler).

2.1.4.1 Auxiliary Frontend

The Auxiliary Frontend creates the IR for the Core Compiler, based on the P4 code as an input. The p4-hlir project[2], which is supported by the P4 organization, it is used to translate the P4 programs into a High Level IR (HLIR). The Auxiliary Frontend is designed as a plug-in framework and allows to aggregate several DSLs. Figure 8 shows the generation of the HLIR from the P4, using the p4-hlir support.

2.1.4.2 Auxiliary Backend

To give a common SDK for the compiler, it is used the Auxiliary Backend. The compiler incorporates the ODP APIs. To create the connection between P4 and ODP abstraction, the the development libraries are necessary. These libraries allow the packet processing inside the dataplane pipeline. This module provides the API auto-generation support, and enables control protocols like Switch Abstraction Interface (SAI)[3], OpenFlow[4], etc.

2.1.4.3 Core Compiler

As the principal part of MACSAD Switch (MacS), the Core Compiler encompasses the Transpiler and Compiler internal modules. With the HLIR generated by the Auxiliary Frontend and the ODP APIs provided by the Auxiliary Backend, it is compiled the MacS. Figure 8 displays the complete Compilation Process.

The **Transpiler** takes the input from the Auxilary Frontend (HLIR) and auto-generates the Datapath Logic codes. In Figure 8 is presented the Transpiler's processing. The Datapath Logic is defined using 'C' language, it is required by the pipeline and used by the Compiler. The Transpiler determines the lookup mechanism, the size, and type of tables that are going to be created, with the resources available on the target. The 'C' codes generated have information from the P4 program, ODP APIs (SDKs), datapath definitions, and helper libraries. The Transpiler allows enabling some code optimizations for example 'Dead Code Elimination' identifying reachability in a dependency graph of parser logic created.

The **Compiler** creates the MacS for the desired target with the 'C' codes generated. With MACSAD is supported the GNU Compiler Collection (GCC) and Low Level Virtual Machine (LLVM) based compiler, supporting multiple targets and optimization tools.

[2] https://github.com/p4lang/p4-hlir
[3] https://github.com/opencomputeproject/SAI
[4] https://www.opennetworking.org/sdn-resources/openflowI

2.1.5 Helper APIs

MACSAD is implemented with the use of a number of APIs (Auxiliary Backend module). These APIs helps the Compiler submodule in the compilation process, covering the Parser and Table functionalities of a P4 program over the ODP SDKs. An example of this Helper APIs or libraries that MACSAD can use is the IPv4 LPM implementation from ODP, adding new features and use cases to the project.

2.2 Related Work

In networking devices, IP lookup forms a bottleneck during the packet forwarding due to the lookup speed unable to deal with the increase in link bandwidth (RAVIKUMAR; MAHAPATRA, 2004).

In PISCES (SHAHHAZ et al., 2016), the packet forwarding and processing are presented in a high-level DSL as P4, compiling down to run on the underlying software switch. The implementation is not protocol specific, which means new features can be added. The researchers compared the performance to projects as OpenvSwitch (OVS) (OVS, 2009) a hardwired hypervisor switch. PISCES program might be optimized about 40 times compared as an equivalent program in Open vSwitch.

The work in (RéTVáRI *et al.*, 2013) shows how to write the IP Forwarding Information Base (FIB) to make forwarding decisions, with basically zero cost on LPM and FIB update. They extend a static entropy compressed FIB representation (based on the labeled tree entropy measure (FERRAGINA *et al.*, 2009)) with optimal lookup. The authors evaluate the algorithm on a Linux kernel prototype and an FPGA implementation. The compressors encode a FIB of more than 440K prefixes to about 100–400 Kbytes of memory, with an increase in lookup throughput and without time reductions with FIB updates. Later, the researchers re-designed the prefix tree, used commonly for IP lookup, to support and update lookups in optimized time. They compare the work with a Trie-based FIB schemes kernel implementation of Patricia trees (SKLOWER, 2001), the trie-based consumes 24 bytes per node and a single IP lookup cost 32 random memory accesses.

Related with the Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond work, CUCKOOSWITCH (ZHOU *et al.*, 2013) presents a Scalable High Performance Ethernet Forwarding. The project is an Ethernet switch, with a design based on software, and created around memory efficiency, high-performance, and highly-concurrent hash table for compact and fast FIB lookup. The presented switch can process 92.22 million packets (minimum sized) per second, configured with eight 10 Gbps Ethernet interfaces on a commodity server. The performance test is also run with a continues maintenance of the forwarding table with one billion entries. The evaluation process is performed in three main steps: (i) they evaluate the forwarding with no switching or FIB lookup involved, these results as a base line for the capacity of the platform using DPDK, (ii) they evaluate the proposed optimizations contribution to the performance of the hash tables and the full system forwarding packets, (iii) the project is compared with other common hash table implementations (INTEL, 2011) (hash table micro-benchmarks and complete system evaluation).

Related to IPv4 and IPv6 forwarding implementations, the work (NIKOLENKO *et al.*, 2016) proposes an abstraction layer able to represent IPv6 FIBs on existing IP and even MPLS infrastructure. Due to most of the forwarding methods that efficiently represent IP-based FIBs do not scale well to IPv6 (larger 128-bit address width) (EATHERTON GEORGE VARGHESE, 2004; RéTVáRI *et al.*, 2013; SRINIVASAN; VARGHESE, 1999). The implementations are common in form of decision tree representation for IPv6 software FIB. The issue with these implementations is that prefix trees are inherently sequential. For this reason, LPM lookup involves multiple consecutive steps, being the total number of steps not optimal for IPv6 (128-bit address). Each step includes separated random access to memory, limiting the total number of levels that the tree can take. For this reason, most of IPv4 FIB trees implementations are not extensible to IPv6 with optimal time and memory requirements. To avoid this trouble, the project appeals to an IPv6 FIB representation on a parallel implementation, lookup classified prefixes into groups, where each group contains all prefixes of the same length and perform the lookup of each group simultaneously.

The work (PONG; TZENG, 2012) shows a distinct LPM lookup scheme to reach concise lookup tables. The project is based on unified hash tables and can handle IPv4 and IPv6 lookup simultaneously. They simplify the table format to earn a better prefix aggregation, also being simplified the implementation process. Due to the hash table implementation, multiple possible buckets are permitted, allowing parallel search over tables during the LPM lookup process. Each lookup takes two cycles on average to complete a lookup and can handle 250 millions of packets per second. A related work (TOBOLA; KOřENEK, 2011) implements a Hash-Tree Bitmap algorithm for fast LPM, also supporting IPv4 and IPv6 lookups. The fast hash implementation allows passing through a limited part of the IP prefix tree. The proposed algorithm uses 16 parallel hash units and two consequent tree Bitmap lookups. The throughput archived by the algorithm reaches 100 Gbps on Virtex 5 FPGA[5] without memory and resources limitations.

For the lookup process, it is commonly used Ternary Content Addressable Memory (TCAM) to facilitate fast IP lookup as it is presented in (HANNA *et al.*, 2011) TCAMs are power-eager, expensive, and not scalable. They perform a forwarding table in tries data structures that are acceded by pipeline. The algorithm proposed is based on a multi-

[5] https://www.xilinx.com/support/documentation/data_sheets/ds100.pdf

bit tree architecture and can reach a throughput of 3.1 Terabits per second. The algorithm archives a better compression ratio, compared with other proposed algorithms (WANG; TZENG, 2006; DEGERMARK *et al.*, 1997; EATHERTONK *et al.*, 2004).

(FIESSLER *et al.*, 2017) presents a Hybrid On-chip Matching combining a highly optimized configuration specialized and thus energy and resource efficient classification circuit with the capability of quickly updated and network packet process at link rate on a Field Programmable Gate Array (FPGA). The evaluation demonstrates that the hybrid implementation benefices the process, resulting in a matching engine that can manage the updates efficiently with a lower hardware resource and power dissipation footprint. The algorithm presented in this work is not restricted to a particular use-case. It can be adapted to an environment where two matching engines with different matching capabilities need to be used.

The project Translator for P4 Switches (T4P4S) (LAKI *et al.*, 2016) is a high speed packet forwarding compiled from protocol independent dataplane specifications, it uses as the auxiliary backend DPDK contrary to MACSAD that uses ODP. T4P4S runs the target independent P4 core in top of a Hardware Abstraction Library (HAL) to improves portability. The developed use cases covered L2 and L3 (IPv4) switch examples.

Primary related projects around SDN, dataplane and IP lookup implementations with IPv4/IPv6 support mentioned above are summarized in Table 3.

Here, we summarize a selected set of related PCAP trace generator solutions. Each tool has their advantages and disadvantages depending on their architecture, including features or supported platforms.

RWS (KNUTSSON, 2014) PCAP generator is based on a simple packet descriptor language. The user defines the header fields for the packets required and feed it to RWS to generate the PCAP. It can also generate invalid packets which is rather uncommon among PCAP generators. An example of an invalid packet can be a TCP packet tunneled inside a Teredo tunnel and sent over GTP-u.

Ostinato (OSTINATO, 2010) is one of the most powerful packet crafter, network traffic generator and analyzer with complete GUI support. It implements most of the common standard protocols to facilitate traffic generation and analysis. With a complex user interface and numerous feature combinations, Ostinato presents a steep learning curve to tackle with, which makes it difficult and time-consuming for users to understand and take advantage of the tool.

Scapy (BIONDI, 2008) is a packet manipulation program with Python interpreter disguised as a DSL. It can create and decode packets of an extended number of protocols. It can send and capture network traffic too. Its extended features also include some basic network tasks (e.g., scanning, trace routing, probing, arpspoof, arp-sk, arping, tcpdump, and tethereal.). Scapy can stand out among competitors with its unique ability to arrange protocol headers in a custom sequence which may not confirm to any protocol logic. This feature allows Scapy to create invalid frames by combining techniques (e.g., VOIP decoding on WEP encrypted channel, and so on.), similar to RWS. Among other features, Scapy allows to set values for all header fields, payload, and padding. Moreover, it allows writing a list of packets to a PCAP file.

With the increasing use of P4 to define the dataplane structure and the efforts to develop complex use cases and applications, the tools to generate traffic, use cases and analyze the code are emerging. Some of the main works for the generations of traces and evaluation are summarized below.

A tool to validate a P4 program by generating test input packets and tables entries is P4pktgen (NÖTZLI *et al.*, 2018). This tool automatically generates test cases for P4 programs using symbolic execution, allowing in parallel to detect bugs in the files. A similar project is the P4app (P4.ORG, 2013). It allows to perform functional tests for a P4 program using BMV2 simple_switch in Mininet environment, being limited to a simulated environment. For a complex test the work P4 benchmark (DANG *et al.*, 2017) test the target compilers by the generation of different P4 applications with variable complexities.

2.3 Summary

This chapter summarized the background of main concepts that sets up the context of the dissertation and the related work and projects that are aligned with this dissertation. This dissertation's approach adds the IPv4 and IPv6 LPM support to MACSAD, creating a complete compiler system with a P4 dataplane base, being different to other approaches that lack the support of IPv6 or do not have a compiler system and a P4 base.

We briefly introduced four main concepts that form an essential part in the dissertation. P4 defining the packets pipeline process, ODP as the APIs and libraries for the dataplane, NFPA as a benchmarking tool, and finally MACSAD to join together P4 and ODP in a common compilation process, creating the software switch image.

Table 3 – Feature comparison list of different IP lookup projects

Project	**IPv4**	**IPv6**	**Compiler**	**P4 Dataplane**	**Target**	**Remarks**
PISCES	Yes	No	Yes	Yes	Software Switch	OVS Based
Compressing IP Forwarding Tables: Towards Entropy Bounds and Beyond	Yes	No	No	No	General-Purpose Processor/ FPGA	Based on prefix tree
CUCKOOSWITCH	Yes	No	Yes	No	General-Purpose Processor	Based on cuckoo hashing
IPv6 Forwarding Tables on IPv4 or MPLS Dataplanes	Yes	Yes	No	No	Limited by DPDK	Parallel prefix trees implementation
Effective Hash-based IPv6 Longest Prefix Match	Yes	Yes	No	No	General-Purpose Processor/ FPGA	Hash-Tree Bitmap architecture
Concise Lookup Tables for IPv4 and IPv6 Longest Prefix Matching in Scalable Routers	Yes	Yes	No	No	Multi-Target	Based on unified hash tables
A Novel Scalable IPv6 Lookup Scheme Using Compressed Pipelined Tries	Yes	Yes	No	No	Multi-Target	Based on trie data structures
T4P4S	Yes	No	Yes	Yes	Multi-Target	X86 & ARMv8 support available
MACSAD (before IPv4/IPv6 Support)	No	No	Yes	Yes	Multi-Target	X86 & ARMv8 support available
MACSAD (after or proposed IPv4/IPv6 support)	Yes	Yes	Yes	Yes	Multi-Target	X86 & ARMv8 support available

3 Design and Implementation of IPv4/IPv6 Longest Prefix Match support

Based on the problems identified in chapter 1 of **Limited support of IPv4 lookup** and **No IPv6 support in ODP**, and in the objectives of **IPv4 and IPv6 LPM support for MACSAD** including the **Controller support**, in the following section, we will describe the implementation choices and the processes embedded in MACSAD IPv4/IPv6 LPM support.

Firstly, we present the IPv4 LPM support implementation with a brief view of the MACSAD and ODP library functions. Secondly, we detail the IPv6 implementation with a complete description of the functions and processes, considering that the base of the application is the same of IPv4 but adding support for a big key size (128 bytes). Thirdly, it is depicted the Controller and following the P4 Code. Finally, we illustrate the LPM diagram flow, summarizing the MACSAD and ODP functions.

3.1 IPv4 LPM support

ODP helper library has implemented an IP lookup algorithm to perform the LPM process. We use the library as support for MACSAD IPv4 lookup implementation. From MacS table control it is called the function that performs the lookup process.

The IP lookup algorithm uses a binary tree to detect the overlap prefix (matching table entries). In a binary prefix tree data structure, each node has at the most two children referred as the left child and the right child. Figure 9 presents the Binary prefix tree description including the relationship through the different tree levels. Splitting the process into different levels and limiting the number of tables, we can reduce memory consumption and maintain a good lookup speed. For the ODP lookup implementation, the number of level 1 entries is 16, and the size of one level 2 and level 3 subtrees are 8. For our implementation we maintain the ODP library definition of levels, fixing the root node in 16, covering the most common prefixes length distributions (RIPE NCC, 2010), and adding two additional levels to complete the 32 bits prefix size. The number of level 2 and level 3 entries (subtrees) per cache cube is 13, and the number of prefix tree nodes per cache cube is 20.

The lookup algorithm uses a structure that stores an entry of the IP prefix table. Because of the leaf pushing, each entry of the table must have either a child entry or a nexthop information. If the child is 0 and the index is valid, this entry has a nexthop

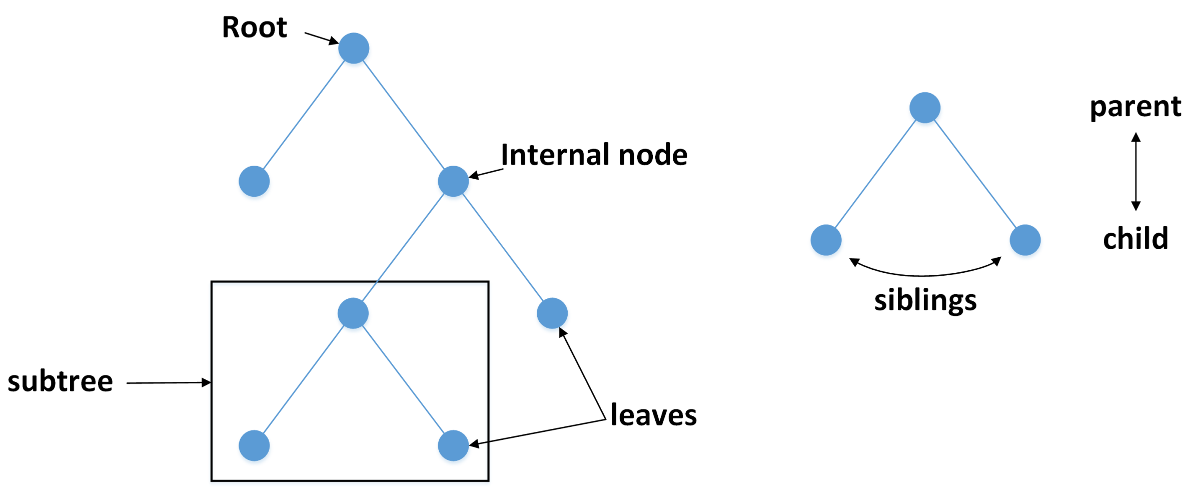


Figure 9 – Binary prefix tree levels. Source: Adapted from (WATERLOO, 2018)

information; the index indicates the buffer that stores the nexthop value. If the child is 1, this entry has a subtree; the index indicates the buffer that stores the subtree.

The LPM lookup has three main processes, the Binary prefix tree, the ODP IP prefix lookup table, and the MACSAD lookup. To implement the lookup process, some basic methods are required depending on the process involved. The methods that we implement are mentioned in the next section along with the relationship between the different lookup process. The complete ODP code implemented can be found in Appendix B.

3.1.1 Binary tree

The process that involves the **Binary prefix tree** is presented in Figure 10[1]. This process is in charge of creating the prefix tree and managing the information that is going to be stored in the specific nodes. The functions involved in the process are:

- ***trie_init***, to initialize the root node of the prefix tree.
- ***trie_destroy***, to destroy the whole prefix tree (recursively).
- ***trie_insert_node***, to insert a new prefix node into the prefix tree. If the node already exists, it is updated the nexthop information, if the node does not exist the target node is created and all nodes along the path from the root to the target node.
- ***trie_delete_node***, to delete a node.
- ***trie_detect_overlap***, to detect the longest overlapping prefix.

[1] For the functions name, it was maintained the ODP format using the term "trie" for the rest of the work it was used the term "tree".

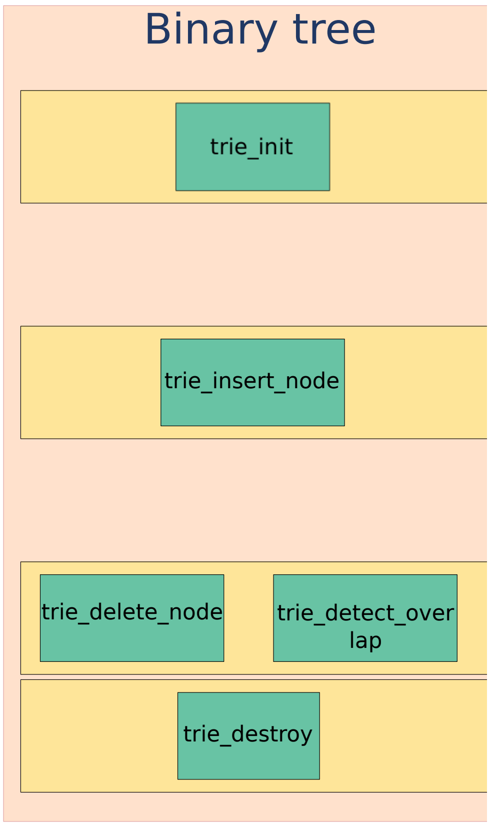


Figure 10 – Binary tree

3.1.2 IP prefix lookup table

To integrate the Binary prefix tree process with the lookup library, the functions presented in Figure 11 are the implemented ones. This process uses the Binary prefix tree methods to originate the tables that are going to store the lookup information. The functions involved are:

- ***odph_iplookup_table_create***, to create a table.
- ***odph_iplookup_table_lookup***, to perform the table lookup.
- ***odph_iplookup_table_destroy***, to destroy the table.
- ***odph_iplookup_table_put_value***, to add a new value into the table.
- ***odph_iplookup_table_get_value***, to get a value stored into the table.
- ***odph_iplookup_table_remove_value***, to remove a entry from the table.

3.1.3 MACSAD lookup

MACSAD uses the IP lookup library, and the functions described above are called. The **MACSAD lookup** process is portrayed in Figure 12. This process sends the data

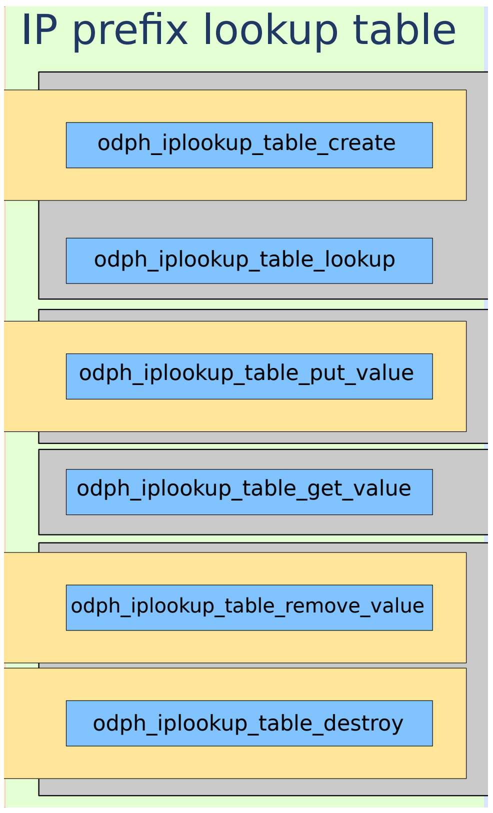

IP prefix lookup table
odph_iplookup_table_create
odph_iplookup_table_lookup
odph_iplookup_table_put_value
odph_iplookup_table_get_value
odph_iplookup_table_remove_value
odph_iplookup_table_destroy

Figure 11 – ODP IP prefix lookup table

to be stored in the tables to the IP prefix lookup table applying the methods and the information described in the section above. The functions involved are:

- ***table__create***, to create and initialize the table.
- ***lpm__add***, to add a LPM value.
- ***lpm__lookup***, to perform the lookup process.
- ***odpc__lookup__tbls__des***, to destroy and remove all the table values.

Figure 13 shows the relationship between the MACSAD lookup functions, the ODP IP prefix lookup table and the Binary prefix tree.

3.2 IPv6 LPM support

3.2.1 Binary tree

For the IPv6 case, we maintain the logic of the ODP IPv4 **Binary prefix tree** implementation, with the additional support for 128 bits addresses. In the following list,

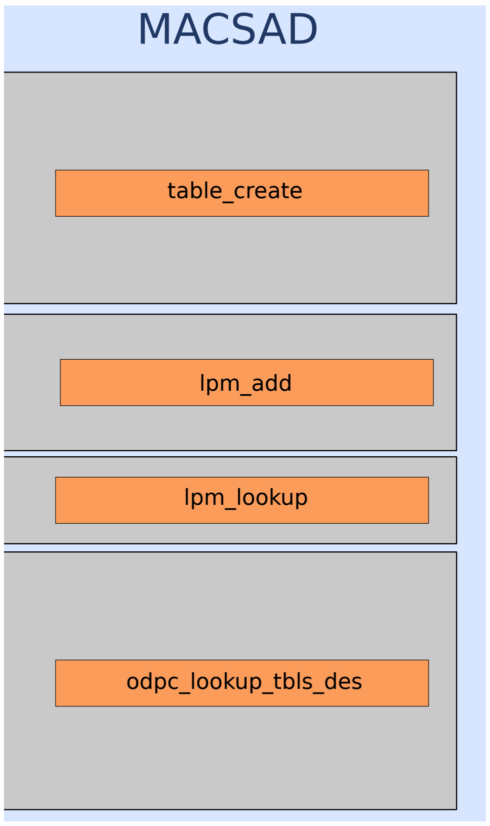


Figure 12 – MACSAD lookup functions

we describe the required parameters for each **Binary prefix tree** functions[2].

- ***trie_init*** function initializes the root node of the prefix tree. With *cache_get_buffer* we get a new buffer from a cache list, if there is no available buffer a new pool will be allocated. In this case, the new buffer will insert into the queue and according to the type of cache will set the initial value of the buffer.

```
trie_init(odph_iplookup_table_impl *tbl){
  trie_node_t *root = NULL;
  odp_buffer_t buffer = cache_get_buffer(tbl, CACHE_TYPE_TRIE);

  if (buffer != ODP_BUFFER_INVALID) {
    root = (trie_node_t *)odp_buffer_addr(buffer);
    root->cidr = 0;
    tbl->trie = root;
    return 0;}
  return -1;
}
```

Listing 3.1 – Binary tree initialization function

[2] In the Listings, it was highlighted in red color the modifications between the existing ODP IPv4 implementation and the new IPv6 implementation.

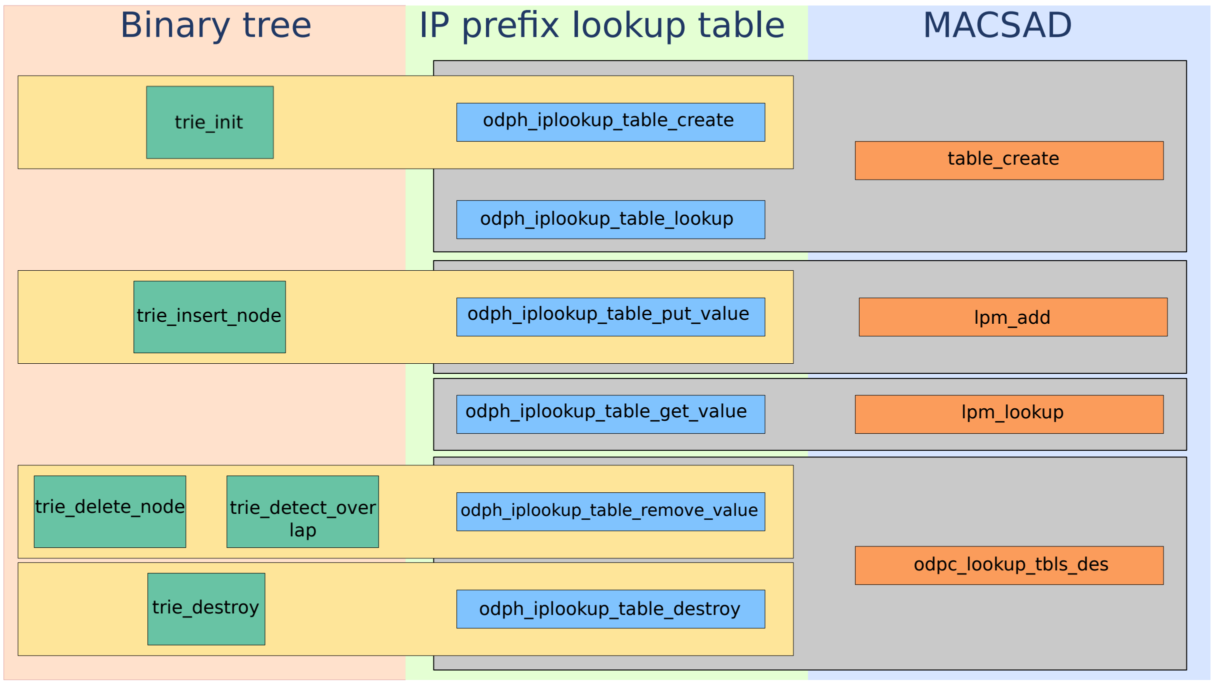

Binary tree
IP prefix lookup table
MACSAD
trie_init
odph_iplookup_table_create
table_create
odph_iplookup_table_lookup
trie_insert_node
odph_iplookup_table_put_value
lpm_add
odph_iplookup_table_get_value
lpm_lookup
trie_delete_node
trie_detect_overlap
odph_iplookup_table_remove_value
odpc_lookup_tbls_des
trie_destroy
odph_iplookup_table_destroy

Figure 13 – Lookup relationship between Binary prefix tree, ODP and MACS

- ***trie_destroy*** function can destroy the whole prefix tree (recursively), verifying the child nodes (left and right).

```
trie_destroy(odph_iplookup_table_impl *tbl, trie_node_t *trie){
  if (trie->left != NULL)
    trie_destroy(tbl, trie->left);
  if (trie->right != NULL)
    trie_destroy(tbl, trie->right);

  odp_queue_enq(
      tbl->free_slots[CACHE_TYPE_TRIE],
      odp_buffer_to_event(trie->buffer));
}
```

Listing 3.2 – Binary tree destroy function

- ***trie_insert_node*** function inserts a new prefix node into the prefix tree. If the node already exists, it is updated the nexthop information, set the return to 0 and the nexthop pointer to INVALID. If the node does not exist, the target node is created and all nodes along the path from the root to the target node, then it is set the return to 0, and the nexthop pointer points to the new buffer. If any error occurs during the process, the return will be set to -1.

```
trie_insert_node(odph_iplookup_table_impl *tbl, trie_node_t *root,
        _uint128_t ip, uint8_t cidr, odp_buffer_t nexthop)
{
  uint8_t level = 0, child;
  odp_buffer_t buf;
  trie_node_t *node = root, *prev = root;
  for (level = 1; level <= cidr; level++) {
    child = WHICH_CHILD(ip, level);
    node = child == 0 ? prev->left : prev->right;
    if (node == NULL) {
      buf = cache_get_buffer(tbl, CACHE_TYPE_TRIE);
      if (buf == ODP_BUFFER_INVALID)
        return -1;
      node = (trie_node_t *)odp_buffer_addr(buf);
      node->cidr = level;
      node->parent = prev;
      if (child == 0)
        prev->left = node;
      else
        prev->right = node;
    }
    prev = node;
  }
  node->nexthop = nexthop;
  return 0;
}
```

Listing 3.3 – Binary tree insert node function

- ***trie_delete_node*** function can be used to delete a node. However, the default prefix (root node) can not be deleted. During the process, while finding the target node all redundant nodes are removed along the path.

```
trie_delete_node(odph_iplookup_table_impl *tbl,
        trie_node_t *root, _uint128_t ip, uint8_t cidr){
  if (root == NULL)
    return -1;
  if (cidr == 0)
    return -1;
  trie_node_t *node = root, *prev = NULL;
  uint8_t level = 1, child = 0;
  odp_buffer_t tmp;
  for (level = 1; level <= cidr; level++) {
    child = WHICH_CHILD(ip, level);
    node = (child == 0) ? node->left : node->right;
    if (node == NULL) {
      return -1;
    }
  }
  node->nexthop = ODP_BUFFER_INVALID;
  for (level = cidr; level > 0; level--) {
    if (
      node->left != NULL || node->right != NULL ||
```

```c
      node->nexthop != ODP_BUFFER_INVALID)
      break;
    child = WHICH_CHILD(ip, level);
    prev = node->parent;
    tmp = node->buffer;
    cache_init_buffer(
        tmp, CACHE_TYPE_TRIE, sizeof(trie_node_t));
    odp_queue_enq(
        tbl->free_slots[CACHE_TYPE_TRIE],
        odp_buffer_to_event(tmp));
    if (child == 0)
      prev->left = NULL;
    else
      prev->right = NULL;
    node = prev;
  }
  return 0;
}
```

Listing 3.4 – Binary tree delete node function

- ***trie_delete_overlap*** function detects the longest overlapping prefix. It is used to remove values from the IPv6 lookup table.

```c
trie_detect_overlap(trie_node_t *trie, _uint128_t ip, uint8_t cidr,
          uint8_t leaf_push, uint8_t *over_cidr,
          odp_buffer_t *over_nexthop){
  uint8_t child = 0;
  uint32_t level, limit = cidr > leaf_push ? leaf_push + 1 : cidr;
  trie_node_t *node = trie, *longest = trie;
  for (level = 1; level < limit; level++) {
    child = WHICH_CHILD(ip, level);
    node = (child == 0) ? node->left : node->right;
    if (node->nexthop != ODP_BUFFER_INVALID)
      longest = node;
  }
  *over_cidr = longest->cidr;
  *over_nexthop = longest->nexthop;
  return 0;
}
```

Listing 3.5 – Binary tree detect overlap function

3.2.2 IP prefix lookup table

To integrate the Binary prefix tree process with the lookup library, the functions presented in Figure 11 are adapted for the IPv6 support. This process uses the Binary prefix tree methods to originate the tables that are going to store the lookup information. The processes involved are:

- ***odph_iplookupv6_table_create*** function creates the lookup table. This function, firstly, is going to check if the parameters are valid. Then it is going to guarantee that there are no existing tables with the same name using the *odph_iplookup_table_lookup* function, after this step the sizes of the different parts of the IP prefix are calculated, as well as the header of this memory block in the implemented structure table where the L1 entries will be arrayed. Finally, it sets the table context, it initializes the cache and the tree.

```
odph_table_t odph_iplookupv6_table_create(const char *name,
          uint32_t p1 ODP_UNUSED,
          uint32_t p2 ODP_UNUSED,
          uint32_t value_size){
  odph_iplookup_table_impl *tbl;
  odp_shm_t shm_tbl;
  odp_queue_t queue;
  odp_queue_param_t qparam;
    .
    .
    .
  tbl = (odph_iplookup_table_impl *)odp_shm_addr(shm_tbl);
  memset(tbl, 0, impl_size + l1_size);

  tbl->l1e = (prefix_entry_t *)(void *)((char *)tbl + impl_size);
  for (i = 0; i < ENTRY_NUM_L1; i++)
    tbl->l1e[i].nexthop = ODP_BUFFER_INVALID;
  snprintf(tbl->name, sizeof(tbl->name), "%s", name);
  tbl->magicword = ODPH_IP_LOOKUP_TABLE_MAGIC_WORD;
  tbl->nexthop_len = value_size;

  for (i = 0; i < 2; i++) {
    tbl->cache_count[i] = 0;
    odp_queue_param_init(&qparam);
    qparam.type = ODP_QUEUE_TYPE_PLAIN;
    sprintf(queue_name, "%s_%d", name, i);
    queue = odp_queue_create(queue_name, &qparam);
    if (queue == ODP_QUEUE_INVALID) {
      ODPH_DBG("failed to create queue");
      cache_destroy(tbl);
      return NULL;}
    tbl->free_slots[i] = queue;
    cache_alloc_new_pool(tbl, i);}
  if (trie_init(tbl) < 0) {
    odp_shm_free(shm_tbl);
    return NULL;}
  return (odph_table_t)tbl;
}
```

Listing 3.6 – Table create function

- ***odph_iplookupv6_table_lookup*** function performs the table lookup to find any match with the received name.

```
odph_iplookupv6_table_lookup(const char *name){
  odph_iplookup_table_impl *tbl = NULL;
  odp_shm_t shm;
  if (name == NULL || strlen(name) >= ODPH_TABLE_NAME_LEN)
    return NULL;
  shm = odp_shm_lookup(name);
  if (shm != ODP_SHM_INVALID)
    tbl = (odph_iplookup_table_impl *)odp_shm_addr(shm);
  if (
    tbl != NULL &&
    tbl->magicword == ODPH_IP_LOOKUP_TABLE_MAGIC_WORD &&
    strcmp(tbl->name, name) == 0)
    return (odph_table_t)tbl;
  return NULL;
}
```

Listing 3.7 – Table lookup function

- ***odph_iplookupv6_table_destroy*** functions to destroy the table. In this function the correct magic word for the IP prefix table will be checked, then they will call the *trie_destroy* function, after this process will be free all the L2 and L3 entries destroying all L3 subtrees of each L2 subtree and then destroying the L2 subtree, finally the cache is destroyed and the memory is cleared.

```
odph_iplookupv6_table_destroy(odph_table_t tbl)
{
  int i, j;
  odph_iplookup_table_impl *impl = NULL;
  prefix_entry_t *subtree = NULL;
  odp_buffer_t *buff1 = NULL, *buff2 = NULL;
  .
  .
  .
  trie_destroy(impl, impl->trie);
  buff1 = ENTRY_BUFF_ARR(impl->l1e);
  for (i = 0; i < ENTRY_NUM_L1; i++) {
    if ((impl->l1e[i]).child == 0)
      continue;
    subtree = (prefix_entry_t *)impl->l1e[i].ptr;
    buff2 = ENTRY_BUFF_ARR(subtree);
    for (j = 0; j < ENTRY_NUM_SUBTREE; j++) {
      if (subtree[j].child == 0)
        continue;
      odp_queue_enq(
          impl->free_slots[CACHE_TYPE_TRIE],
          odp_buffer_to_event(buff2[j]));}
    odp_queue_enq(
        impl->free_slots[CACHE_TYPE_TRIE],
        odp_buffer_to_event(buff1[i]));}
  cache_destroy(impl);
  odp_shm_free(odp_shm_lookup(impl->name));
  return 0;
}
```

Listing 3.8 – Table destroy function

- ***odph_iplookupv6_table_put_value*** functions to add new values into the table. This function is going to verify if the IP, Key, Prefix, and value are not null. The IP will be parsed once its value is obtained. After this process, are set the L1 entries, the values are inserted into the tree depending on the value of the prefix. If it is less than the root prefix (16 bits) it will be inserted using the function *prefix_insert_into_lx*; this function will handle the last 8 bits when it is inserting into the table. It is used the function *prefix_insert_iter* if the prefix is higher than the root and it will be used to insert groups of octets.

```
int odph_iplookupv6_table_put_value(odph_table_t tbl, void *key, void
   *value){
  odph_iplookup_table_impl *impl = (void *)tbl;
  odph_iplookupv6_prefix_t *prefix = (odph_iplookupv6_prefix_t *)key;
  prefix_entry_t *l1e = NULL;
  odp_buffer_t nexthop;
  int ret = 0;
  _uint128_t lkp_ip = 0;
  if ((tbl == NULL) || (key == NULL) || (value == NULL))
    return -1;
  nexthop = *((odp_buffer_t *)value);
  if (prefix->cidr == 0)
    return -1;
  ret = odph_ipv6_addr_parse(&lkp_ip, "ffff:ffff:ffff:ffff:ffff:ffff:
   ffff:ffff");
  if (ret < 0) {
    printf("Failed to get IPv6 addr from str\n");
    return -1;}
  prefix->ip = prefix->ip & (lkp_ip << (IP_LENGTH - prefix->cidr));
  ret = trie_insert_node(
      impl, impl->trie,
        prefix->ip, prefix->cidr, nexthop);
  if (ret < 0) {
    ODPH_DBG("failed to insert into trie\n");
    return -1;}
  l1e = &impl->l1e[prefix->ip >> 112];
  odp_buffer_t *buff = ENTRY_BUFF_ARR(impl->l1e) + (prefix->ip >> 112)
    ;
  if (prefix->cidr <= 16) {
    ret = prefix_insert_into_lx(
        impl, l1e, prefix->cidr, nexthop, 16);
  } else {
    ret = prefix_insert_iter(impl, l1e, buff,
        ((prefix->ip) << 112), prefix->cidr - 16,
        nexthop, 24, 2);}
  return ret;
}
```

Listing 3.9 – Table Put value function

- ***odph_iplookupv6_table_get_value*** function gets a value stored into the table. This function is going to verify if the table, Key, and buffer are not null, then the L1 entry will be obtained and will start searching in the tree, the resulting data will be copied, and if there is not a match will return only match the default prefix and an ODP buffer invalid.

```
int odph_iplookupv6_table_get_value(odph_table_t tbl, void *key,
          void *buffer ODP_UNUSED,
          uint32_t buffer_size ODP_UNUSED){
  odph_iplookup_table_impl *impl = (void *)tbl;
  _uint128_t ip;
  prefix_entry_t *entry;
  odp_buffer_t *buff = (odp_buffer_t *)buffer;

  if ((tbl == NULL) || (key == NULL) || (buffer == NULL))
    return -EINVAL;

  ip = *((_uint128_t *)key);
  entry = &impl->l1e[ip >> 112];
  if (entry == NULL) {
    ODPH_DBG("failed to get L1 entry.\n");
    return -1;}
  ip <<= 112;
  while (entry->child) {
    entry = (prefix_entry_t *)entry->ptr;
    entry += ip >> 24;
    ip <<= 8;}
  if (entry->nexthop == ODP_BUFFER_INVALID) {
    printf("only match the default prefix\n");
    *buff = ODP_BUFFER_INVALID;
  } else {
    *buff = entry->nexthop;}
  return 0;
}
```

Listing 3.10 – Table Get Value function

- ***odph_iplookupv6_table_remove_value*** functions to remove a entry from the table. This function will start checking if the table and the key (including the prefix) are not null, then it detects if there is a prefix overlap using the *trie_detect_overlap* function, if the prefix is less than the root it is used the function *prefix_delete_lx*. If the return of this the function is equal to 1, the next 2^8 entries will equal to over_cidr and over_nexthop. In this case, will be not pushed the over_cidr and over_nexthop to the next level. In the other case when the prefix is higher than the root, it is used the function *prefix_delete_iter* destroying the subtrees, after this, it checks if it can recycle the entry. An entry can be recycled due to two reasons: all children of the entry are the same, or all children of the entry have a prefix smaller than the level bottom bound. At the end the function will finish with the *trie_delete_node*.

```
int odph_iplookupv6_table_remove_value(odph_table_t tbl, void *key) {
  odph_iplookup_table_impl *impl = (void *)tbl;
  odph_iplookupv6_prefix_t *prefix = (odph_iplookupv6_prefix_t *)key;
  _uint128_t ip;
  uint8_t cidr;

  if ((tbl == NULL) || (key == NULL))
    return -EINVAL;
  ip    = prefix->ip;
  cidr  = prefix->cidr;
  if (cidr == 0)
    return -EINVAL;
  prefix_entry_t *entry = &impl->l1e[ip >> 112];
  odp_buffer_t *buff = ENTRY_BUFF_ARR(impl->l1e) + (ip >> 112);
  uint8_t over_cidr, ret;
  odp_buffer_t over_nexthop;
  trie_detect_overlap(
      impl->trie, ip, cidr, 16, &over_cidr, &over_nexthop);
  if (cidr <= 16) {
    prefix_delete_lx(
      impl, entry, buff, cidr, over_cidr, over_nexthop, 16);
  } else {
    prefix_entry_t *ne = (prefix_entry_t *)entry->ptr;
    odp_buffer_t *nbuff = ENTRY_BUFF_ARR(ne);
    ne += ((_uint128_t)(ip << 112) >> 24);
    nbuff += ((_uint128_t)(ip << 112) >> 24);
    ret = prefix_delete_iter(impl, ne, nbuff, ip, cidr - 16, 24, 2);
    if (ret && can_recycle(entry, 16)) {
      /* destroy subtree */
      cache_init_buffer(
        *buff, CACHE_TYPE_SUBTREE,
        sizeof(prefix_entry_t) * ENTRY_NUM_SUBTREE);
      odp_queue_enq(
        impl->free_slots[CACHE_TYPE_SUBTREE],
        odp_buffer_to_event(*buff));
      entry->child = 0;
      entry->cidr = over_cidr;
      entry->nexthop = over_nexthop;}}
  return trie_delete_node(impl, impl->trie, ip, cidr);
}
```

Listing 3.11 – Table Remove value function

In addition to the IP prefix lookup table and binary tree, a function is necessary to parse the IP address. The *odph_ipv6_addr_parse* function reads the IP as a string, and it parses into a vector with the 128 bits. The received string segments it into four sections of 32 bits to parse the IP, and then all that section will be joined at the final IP address variable.

```
int odph_ipv6_addr_parse(_uint128_t *ip_addr, const char *str){
  unsigned byte[ODPH_IPV6ADDR_LEN];
  int i;
  _uint128_t p_ip1, p_ip2, p_ip3, p_ip4;
  memset(byte, 0, sizeof(byte));
  if (sscanf(str, "%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x
   %02x:%02x%02x:%02x%02x",
      &byte[0], &byte[1], &byte[2], &byte[3], &byte[4], &byte[5], &
   byte[6], &byte[7], &byte[8], &byte[9],
      &byte[10], &byte[11], &byte[12], &byte[13], &byte[14], &byte
   [15]) != ODPH_IPV6ADDR_LEN)
    return -1;
  for (i = 0; i < ODPH_IPV6ADDR_LEN; i++)
    if (byte[i] > 255)
      return -1;

  p_ip1 = byte[0] << 24 | byte[1] << 16 | byte[2] << 8 | byte[3];
  p_ip2 = byte[4] << 24 | byte[5] << 16 | byte[6] << 8 | byte[7];
  p_ip3 = byte[8] << 24 | byte[9] << 16 | byte[10] << 8 | byte[11];
  p_ip4 = byte[12] << 24 | byte[13] << 16 | byte[14] << 8 | byte[15];
  *ip_addr = p_ip1 << 96 | p_ip2 << 64 |  p_ip3 << 32 | p_ip4;

  return 0;
}
```

Listing 3.12 – ODP IPv6 Parse

3.2.3 MACSAD lookup

For the LPM MACSAD process, the necessary functions are in charge of creating the tables, add values and get values. MACSAD parses the packets and receives the information of IP to perform the LPM match. The controller reads the entries from the trace file and adds into the tables with the prefix information.

- ***table_create*** function will create the match table; it can be an exact table or an LPM table. It is selected depending on the size of the key if it is IPv4 or IPv6. After this process a table LPM lookup will be performed, if the result is not null the table is destroyed (*odph_iplookup_table_destroy*) then the table is created using the ODP function *odph_iplookup_table_create*.

```
case LOOKUP_LPM:
     snprintf(name, sizeof(name), "%s_lpm_%d_%d", t->name, socketid,
    replica_id);
     if(t->key_size <= 5){ //IPV4
         if ((tbl = odph_iplookup_table_lookup(name)) != NULL){
             odph_iplookup_table_destroy(tbl);}
         // name, capacity, key_size, value size
         tbl = odph_iplookup_table_create(name, 2, t->key_size, t->
    val_size);
         if(tbl == NULL) {
             debug("   :: Table %s creation fail\n", name);
             exit(0);}
         create_ext_table(t, tbl, socketid);
     else if(t->key_size <= 17){ //IPV6
         if ((tbl = odph_iplookupv6_table_lookup(name)) != NULL){
             odph_iplookupv6_table_destroy(tbl);}
         tbl = odph_iplookupv6_table_create(name, 2, t->key_size, t->
    val_size);
         if(tbl == NULL) {
             debug("   :: Table %s ipv6 creation fail\n", name);
             exit(0);}
         create_ext_table(t, tbl, socketid);
```

Listing 3.13 – MACSAD Table Create

- ***lpm_add*** function will add entries inside the LPM table. As a first step, it is used the Key to verify if it is IPv4 or IPv6. In both cases, the logic is the same, just the ODP functions and the size of the variables will be change to the corresponding case. In this function, the IP is validated and then it is parsed to be added to the LPM table with *odph_iplookupv6_table_put_value*.

```
else if(t->key_size <= 17){
    key[16] = depth;
    unsigned byte[ODPH_IPV6ADDR_LEN+1];
    odph_iplookupv6_prefix_t prefix2;
    for (int i = 0; i < ODPH_IPV6ADDR_LEN; i++)
        if (key[i] > 255)
            return;
    .
    .
    .
    prefix2.ip = p_ip1 << 96 | p_ip2 << 64 |  p_ip3 << 32 | p_ip4;
    prefix2.cidr = 64;
    ext->content[ext->size] = copy_to_socket(value, t->val_size+sizeof
   (int), t->socketid);
    value3 = malloc(t->val_size);
    memcpy(value3, value, t->val_size);
    ret = odph_iplookupv6_table_put_value(ext->odp_table, &prefix2, &
   value3);
    ext->size++;
    if (ret == -1) {
        exit(EXIT_FAILURE);}
```

Listing 3.14 – MACSAD Table Add

- ***lpm_lookup*** function will be in charge to perform the table get. This function matches the Key to detect if it is IPv4 or IPv6. The IP is parsed, and the lookup process is performed using the *odph_iplookupv6_table_get_value*.

```
else if(t->key_size <= 17){
    unsigned byte[ODPH_IPV6ADDR_LEN+1];
    memset(byte, 0, sizeof(byte));
    .
    .
    .
    lkp_ip2 = p_ip1 << 96 | p_ip2 << 64 |  p_ip3 << 32 | p_ip4;
    ret = odph_iplookupv6_table_get_value(ext->odp_table, &lkp_ip2, &
   result, 0);
    if (ret < 0) {
        return t->default_val;
```

Listing 3.15 – MACSAD Table lookup

3.3 Controller

The controller in an SDN architecture is in charge of controlling the dataplane elements. In the case of MACSAD the controller is going to handle the actions and the information to be filled in the tables defined at the P4 code. The MACSAD Controller code implemented can be found in Appendix C. For the use case of IPv4 and IPv6 the tables defined at the P4 code are: *ipv6_fib_lpm* and *sendout*.

The controller can read and analyze from an input file (Table Trace) to fill the information in the tables. The Table Trace files have a specific format and information depending on the MACSAD use case. In the case of the IPv4 and IPv6 use cases, the information required is destination IP, MAC, and port addresses. The controller will parse the Table Trace file and extract the corresponding information, creating the variables that are going to be used to fill the values in the tables.

```c
if (11 == sscanf(line, "%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x %x:%x:%x:%x:%x:%x %d",
        &values_ip[0], &values_ip[1], &values_ip[2], &values_ip[3],
        &values_ip[4], &values_ip[5], &values_ip[6], &values_ip[7],
        &values_ip[8], &values_ip[9], &values_ip[10], &values_ip[11],
        &values_ip[12], &values_ip[13], &values_ip[14], &values_ip[15],
        &values[0], &values[1], &values[2],
        &values[3], &values[4], &values[5], &port) ){
    if (mac_count==MAX_MACS-1){
      break;}
    ++mac_count;
    for( i = 0; i < 6; ++i )
      macs[mac_count][i] = (uint8_t) values[i];
    for( i = 0; i < 16; ++i )
      ips[mac_count][i] = (uint8_t) values_ip[i];
    portmap[mac_count] = (uint8_t) port;
  } else {
    fclose(f);return -1;
  }
```

Listing 3.16 – Controller Parse trace file

To fill the table with information in the P4 code, the controller uses the functions *fill_ipv6_fib_lpm_table* and *fill_sendout_table* to fill the information from the Trace file into the tables. The *fill_ipv6_fib_lpm_table* function defines the default actions of the P4 table *ipv6_fib_lpm* to *fib_hit_nexthop*, being necessary the parameters of IP, Port, and destination MAC address. Additionally, this function sends the IP to the ODP LPM table. The *fill_sendout_table* function sets the default action to the P4 table *sendout* to *rewrite_src_mac* being necessary to pass the Port and the defined MAC address. Additionally, a sleep of 1000us between every addition it is necessary to let the ODP functions add the entries without missing any information.

```
void init() {
  int i;
  uint8_t smac[6] = {0xd0, 0x69, 0x0f, 0xa8, 0x39, 0x90};
  for (i=0;i<=mac_count;++i){
    fill_ipv6_fib_lpm_table(ips[i], portmap[i], macs[i]);
    fill_sendout_table(portmap[i], smac);
          usleep(1000);}
}
```

Listing 3.17 – Controller initialization function

3.4 P4 Code

In a P4 code, the headers of the packets are defined at the beginning of the code, in the case of IPv4 and IPv6 the headers are Ethernet and IP. The complete P4 code implemented can be found in Appendix D.

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

header_type ipv6_t {
    fields {
        version : 4;
        trafficClass : 8;
        flowLabel : 20;
        payloadLen : 16;
        nextHdr : 8;
        hopLimit : 8;
        srcAddr : 128;
        dstAddr : 128;}}
```

Listing 3.18 – IPv6 P4 Headers

The necessary tables to forward the packets are the *ipv6_fib_lpm* and *sendout*.

- ***ipv6_fib_lpm*** is going to be in charge to call the LPM function when a valid IPv6 packet arrives. This function reads the destination address of the packet and matches with LPM. The actions if it is found a match are the *fib_hit_nexthop* that modifies the Ethernet designation address with the nexthop information, updates the egress port to the corresponding one, and reduces the hopLimit in one, the other action of the function it is *on_miss* that discards the packet.

- ***sendout*** table is going to read the egress port and perform an exact match, if a match is found, the action will be *rewrite_src_mac* updating the source MAC address of the forwarded packet.

```
action on_miss() {}

action fib_hit_nexthop(dmac, port) {
  modify_field(ethernet.dstAddr, dmac);
  modify_field(standard_metadata.egress_port, port);
  add_to_field(ipv6.hopLimit, -1);}

table ipv6_fib_lpm {
  reads {
    ipv6.dstAddr : lpm;}
  actions {
    fib_hit_nexthop;
    on_miss;}
  size : 512;}

action rewrite_src_mac(smac) {
  modify_field(ethernet.srcAddr, smac);}

table sendout {
  reads {
    standard_metadata.egress_port : exact;}
  actions {
    on_miss;
    rewrite_src_mac;}
  size : 512;}

control ingress {
  apply(ipv6_fib_lpm);
  apply(sendout);}
```

Listing 3.19 – IPv6 P4 Tables and Actions

The control ingress is the place where the flow is going to be defined, fist it is going to apply the *ipv6_fib_lpm* table and then the *sendout* table. In the use case of MACSAD all the tables are defined at the ingress.

```
control ingress {
  apply(ipv6_fib_lpm);
  apply(sendout);}

control egress {}
```

Listing 3.20 – IPv6 P4 Control Ingress and Egress

In Table 4, we compare the use case complexity of the L2 and L3 forwarding with IPv4 and IPv6. The *Parsing* refers to the headers and fields that are parsed at the P4 code, in the case of L3 forwarding the headers are two, Ethernet and IP and the number of fields depend on the protocol (IPv4 or IPv6). *Processing* contains the information of the tables defined, the IPv4 and IPv6 use cases have two, the *ipv6_fib_lpm* and *sendout*. In *Packet Modification* are mentioned the headers that are added or removed from the final packet. *Metadata* is the local information that passes through the tables, in MACSAD is used to pass the egress port of the switch. In *Action Complexity* are summarized the fields and expressions that are modified, in the L3 cases, the fields are Ethernet source and destination address, the Time to Live (TTL) (hop limit for IPv6), and the metadata. The *Lookups* can be Hash (exact) or LPM, the exact match is used to find the egress port and the LPM for the IP lookup.

Comparing the use cases, the IPv6 has 1 LPM table with the key size of 128 bits being higher than the IPv4 case. The Packets fields parsed in the IPv6 are also more than the parsed in the IPv4. On the other hand, the L2 forwarding case has one header with three fields and two hash lookups of 48 bits.

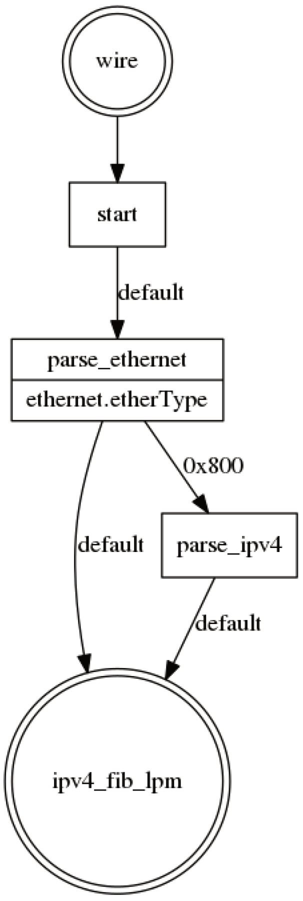
The flow diagram of the P4 code (Generated using P4 Graphs[3]) is speared in the Parse and the Table flows. The Parse flow shows the logic while parsing a packet. In Figure 14b, it is presented the IPv6 Parse, when a packet arrives, the first step is to parse the Ethernet header if the eherType is IPv6 then the IP header is parsed. After this process the tables are executed. The Table flow shows the pipeline of the ingress and egress, in Figure 15b can be seen the order of the applied tables. First, the *ipv6_fib_lpm* and finally the *sendout* table. After this step, the packet is sent to the egress to be forward.

3.5 LPM Flow Diagrams

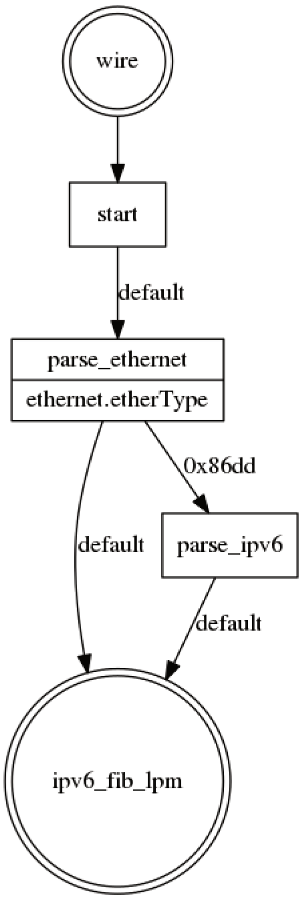
In this subsection, we describe the flow diagram of the LPM implementation containing the relationship between the implemented function of the binary tree, ODP lookup, and MACSAD lookup. Additionally, it is described the flow diagram of the P4 implementation.

In Figure 16, the main OPD LPM lookup flow is observed, when an ODP program starts, the first steps are to initialize the ODP global and local variables (Internal functions of ODP), with this function are set parameters as the distribution of cores, the RX and TX ques, the memory blocs to be used. The tables (Figure 17) are created after this step. Once the tables are created, if new action is requested the flow will continue, the table actions (Figure 18) as described in the previous subsections, the Table Put values, Tables Get Values, Table Remove Value, and the Tables Destroy (Figure 17).

[3] <https://github.com/p4lang/p4-hlir/blob/master/bin/p4-graphs>

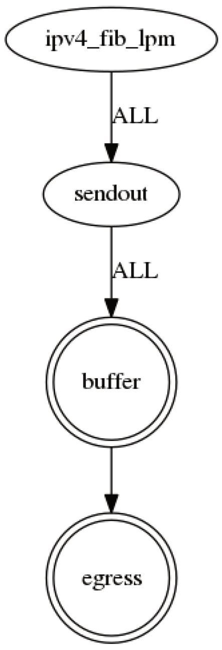


(a) IPv4 Parse

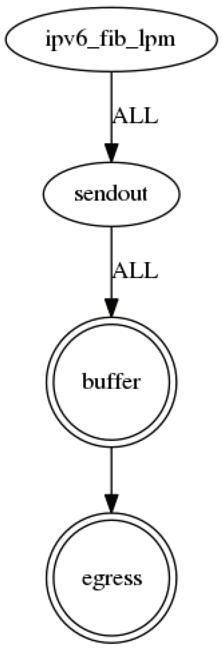


(b) IPv6 Parse

Figure 14 – L3-FWD Parse Flow



(a) IPv4 Table



(b) IPv6 Table

Figure 15 – L3-FWD Table Flow

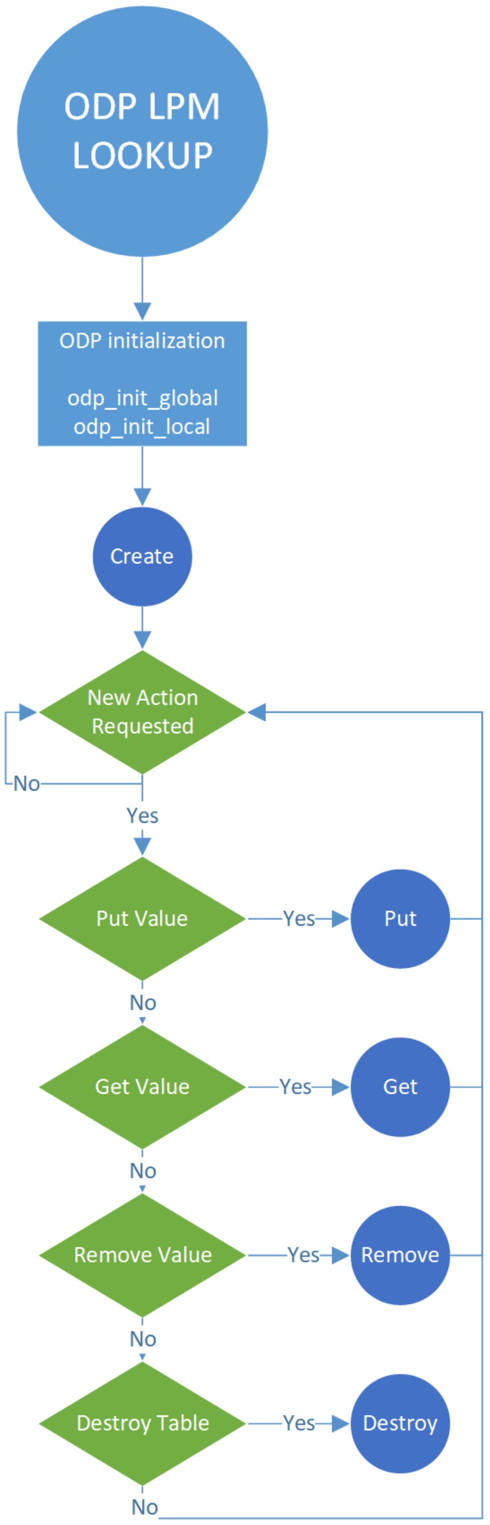


Figure 16 – ODP LPM main flow

The MACSAD LPM implementation flow diagram is described in Figure 19. Following the same process as described above when an ODP program starts, the OPD is initialized, then the tables are created (Figure 20). MACSAD has two types of tables, the Lookup Exact, and the Lookup LPM. The exact match for the L2 information is used (Source and destination MAC address), and available types are the Cuckoo and the Hash table.

MACSAD uses a key to pass the IP and prefix values to the functions. For IPv4 the key size is 5, where the first 4 are the IP, and the last one is the prefix that will be used for the LPM table. For the IPv6 case, the size of the key is 17 where the first 16 are the IP and the last one the prefix. For the LPM lookup tables, the code will match the size of the key; if it is 5, it will be an IPv4 address creating the corresponding table, if the key is 17, the table created will be an IPv6 type. For the MACSAD use case, one table is created for the LPM values. The implemented table actions (Figure 21) are the Table Add and Table Get. The Table Add matches the size of the key with the same logic of the table create and the corresponding IPv4 and IPv6 ODP put value functions call (*odph_iplookup_table_put_value* or *odph_iplookupv6_table_put_value*). The Table Get uses the same logic for the size of the key and depending on the size will be called the the ODP get value functions (*odph_iplookup_table_get_value* or *odph_iplookupv6_table_get_value*).

3.6 Summary

In this chapter we described the IPv4 LPM support implementation in ODP including our MACSAD support for the LPM library, we detailed the IPv6 implementation including our LPM helper library in ODP and our MACSAD support that coverage the controller and the P4 Code design. In the ODP implementation is described the different functions that are part of the LPM process, presenting our IPv6 LPM implementation and comparing it with the IPv4 library developed by ODP. We illustrated the complete workflow of the LPM implementation including the MACSAD and ODP library parts, having a comprehensive view of the process and how the different projects are integrated, also comparing the use case complexity of the L2 and L3 forwarding with IPv4 and IPv6.

In the next chapter, we are going to present our Packet Generator Tool BB-Gen, that will support us in the evaluation of the implemented functions and libraries, allowing us to generate the different packet traces for the performances test.

Table 4 – P4 Use Case Complexity

		L2-FWD	L3-FWD (IPv4)	L3-FWD (IPv6)
		P4_14	P4_14	P4_14
Parsing	#Packet headers	1	2	2
	#Packet fields	3	13	19
	#Branches in parse graph	1	2	2
Processing	#Tables (no dep)	2	2	2
	Depth of pipeline	2	2	2
	Checksum on/off	off	off	off
	Table size	8192	512	512
State Accesses	#Writes to different register	0	0	0
	#Writes to same register	0	0	0
	#Reads to different register	0	0	0
	#Reads to same register	0	0	0
Packet Modification	#Header adds	0	0	0
	#Header removes	0	0	0
Metadata	#Metadatas	1	1	1
	Metadata size(bits)	9	9	9
Action Complexity	#Field writes	2	4	4
	#Arithmetic expressions	0	0	0
	#Boolean expressions	0	0	0
Lookups	#Hash_lookups [key_length(bits)]	2 [48]	1 [9]	1 [9]
	#LPM [key_length(bits)]	0	1 [32]	1 [128]

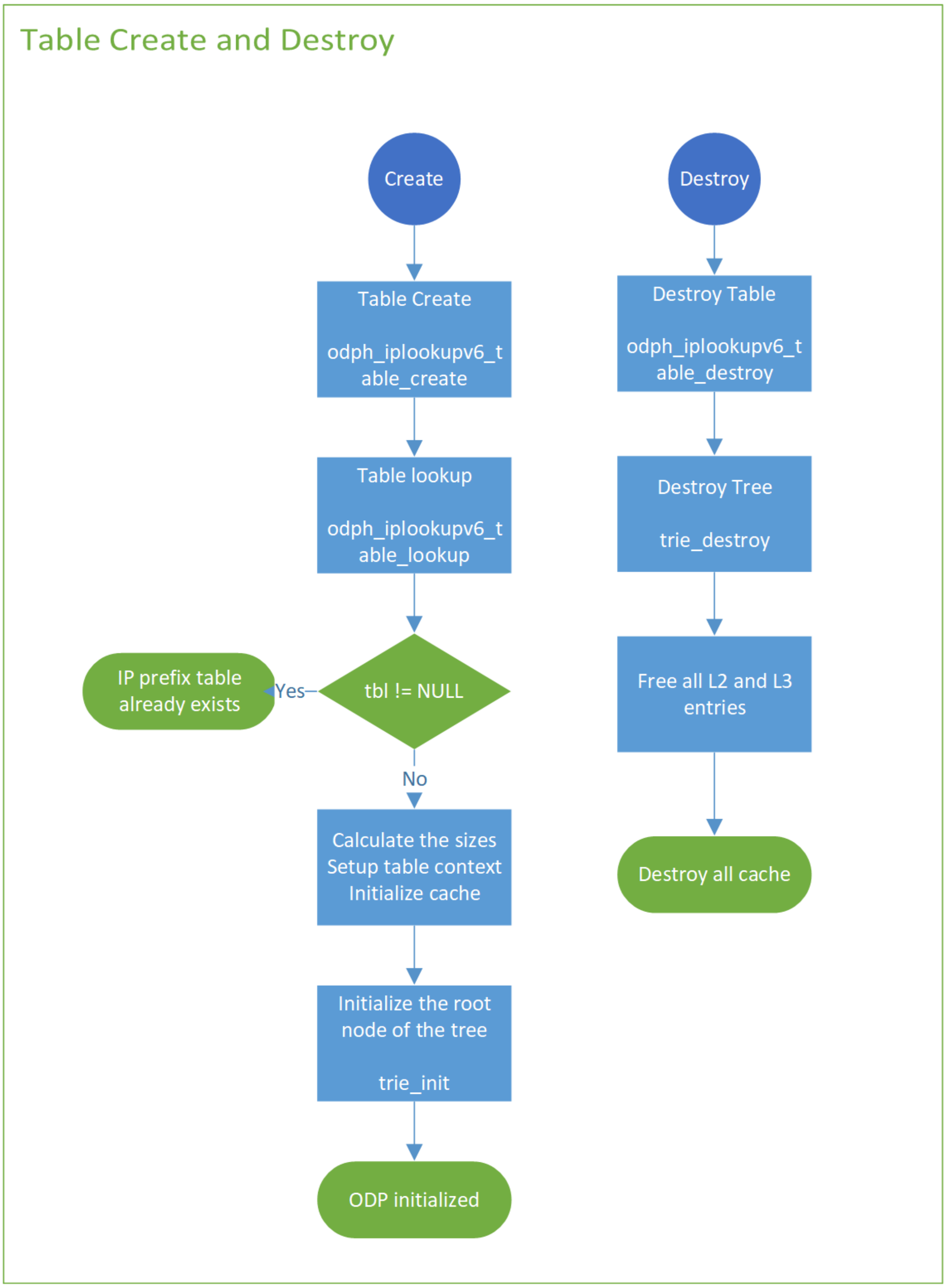


Figure 17 – ODP Table Create and Destroy

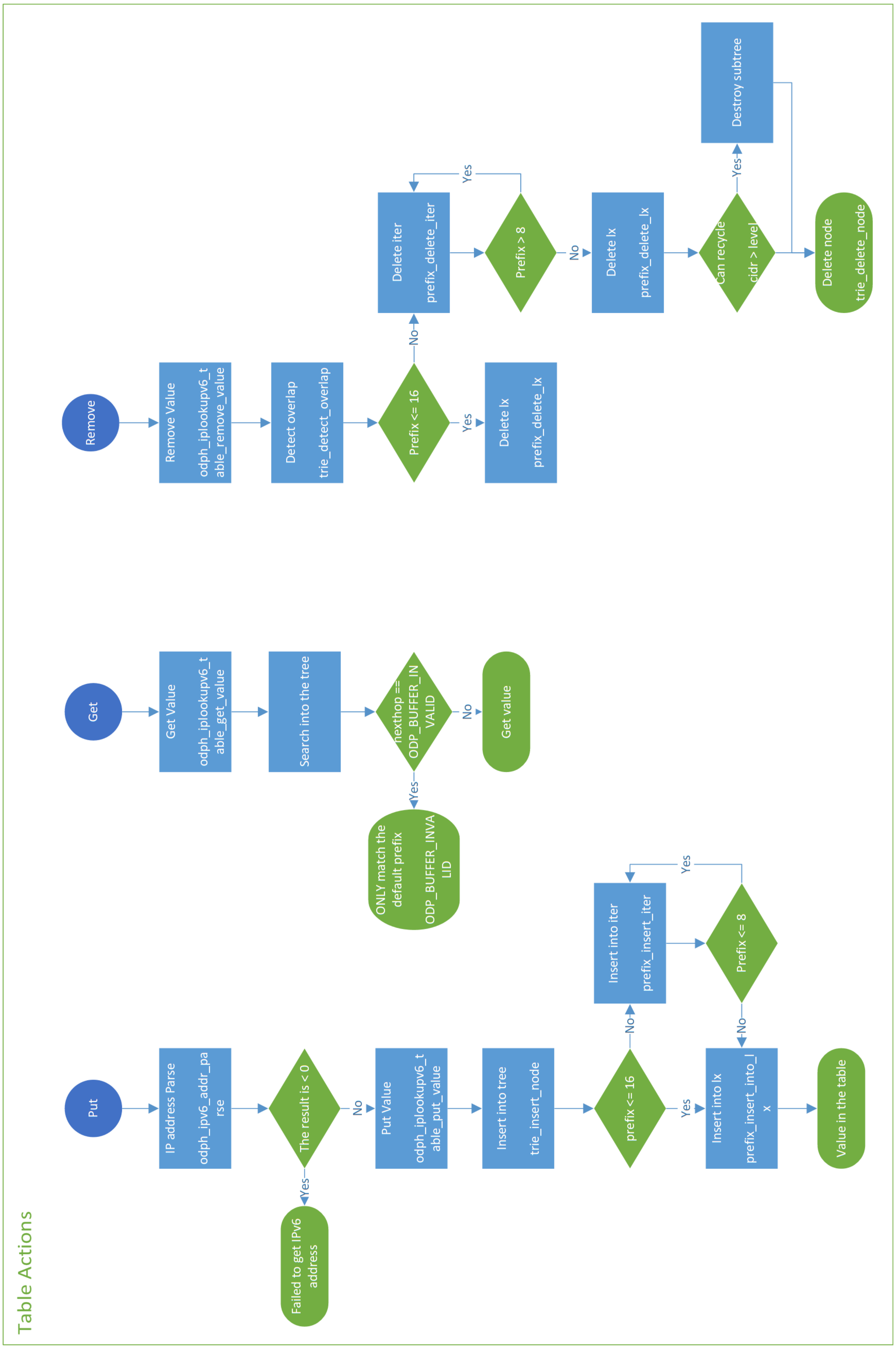


Figure 18 – ODP Table Actions

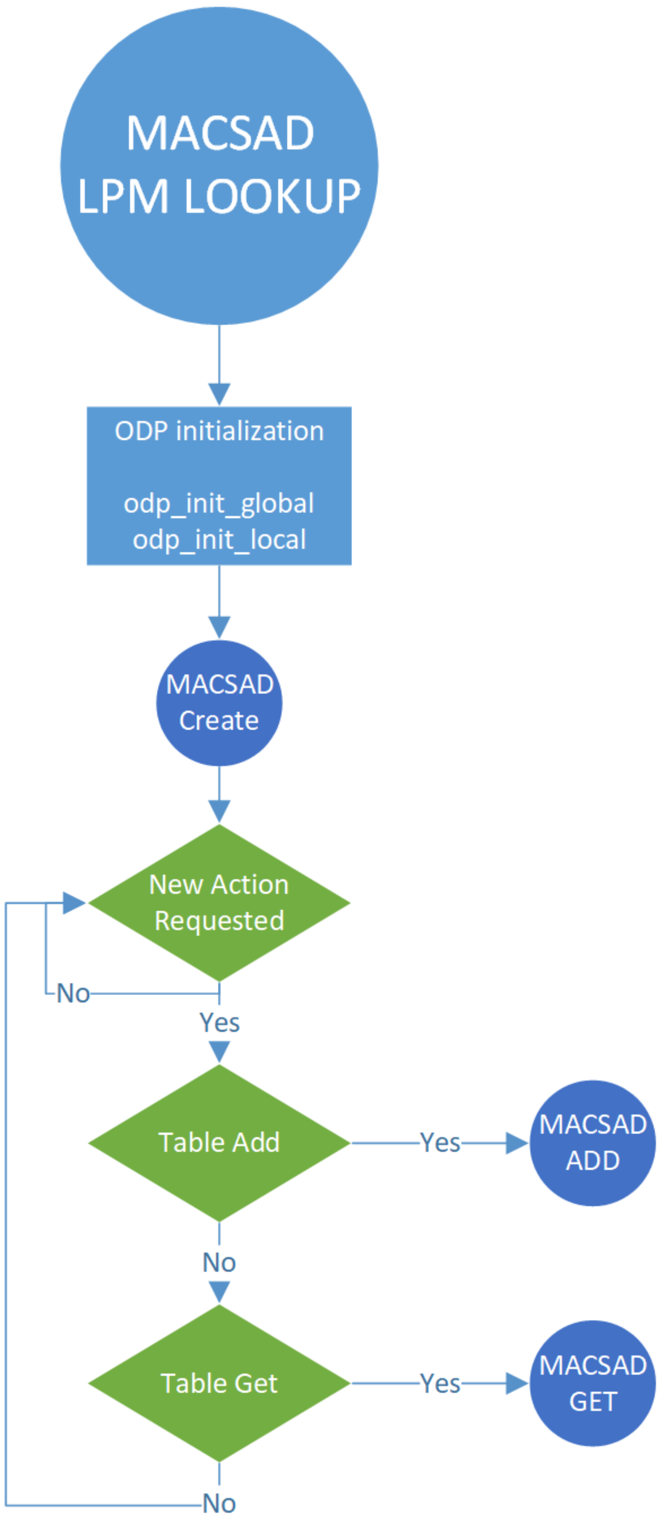


Figure 19 – ODP LPM main flow

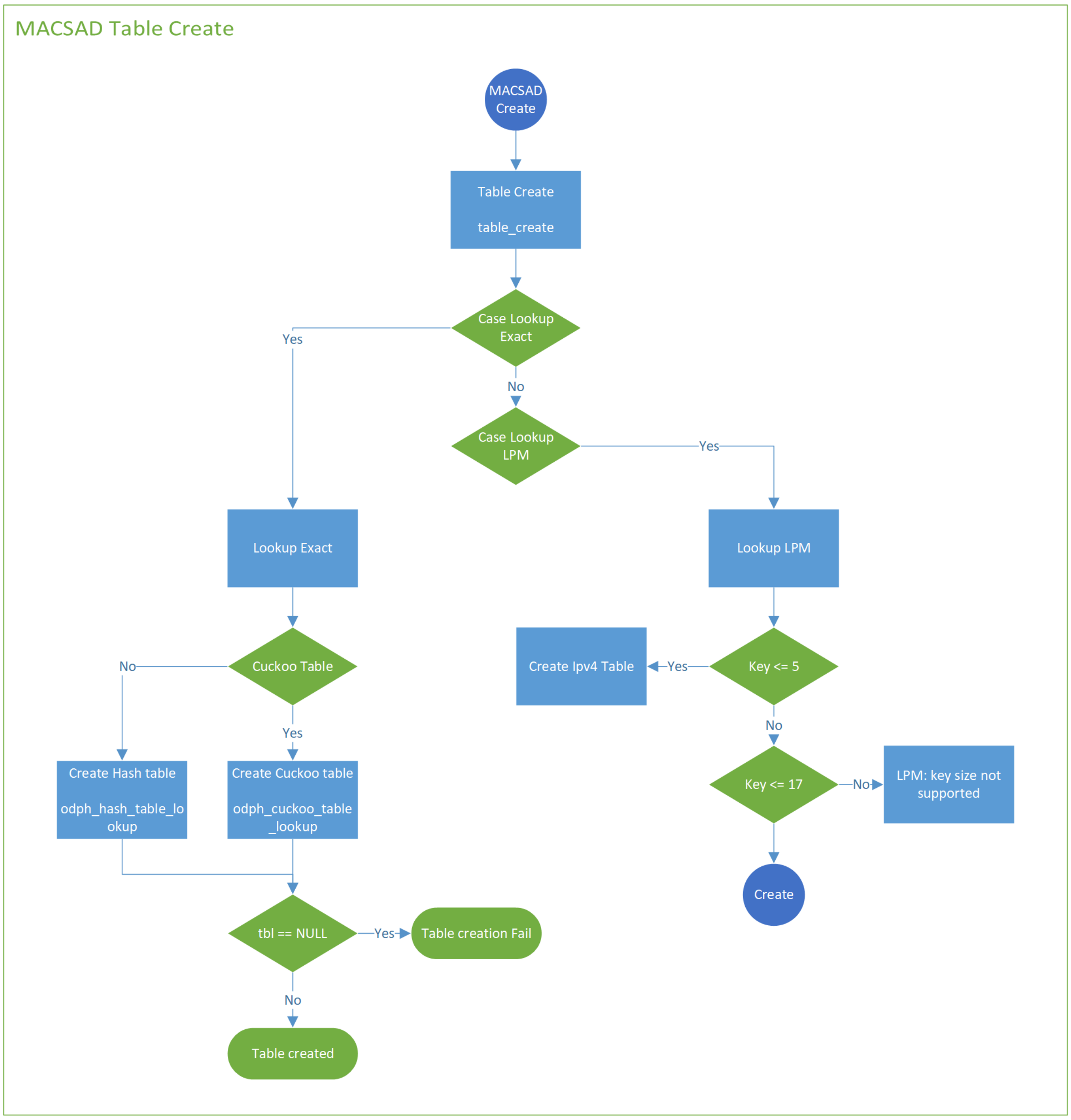


Figure 20 – ODP Table Create

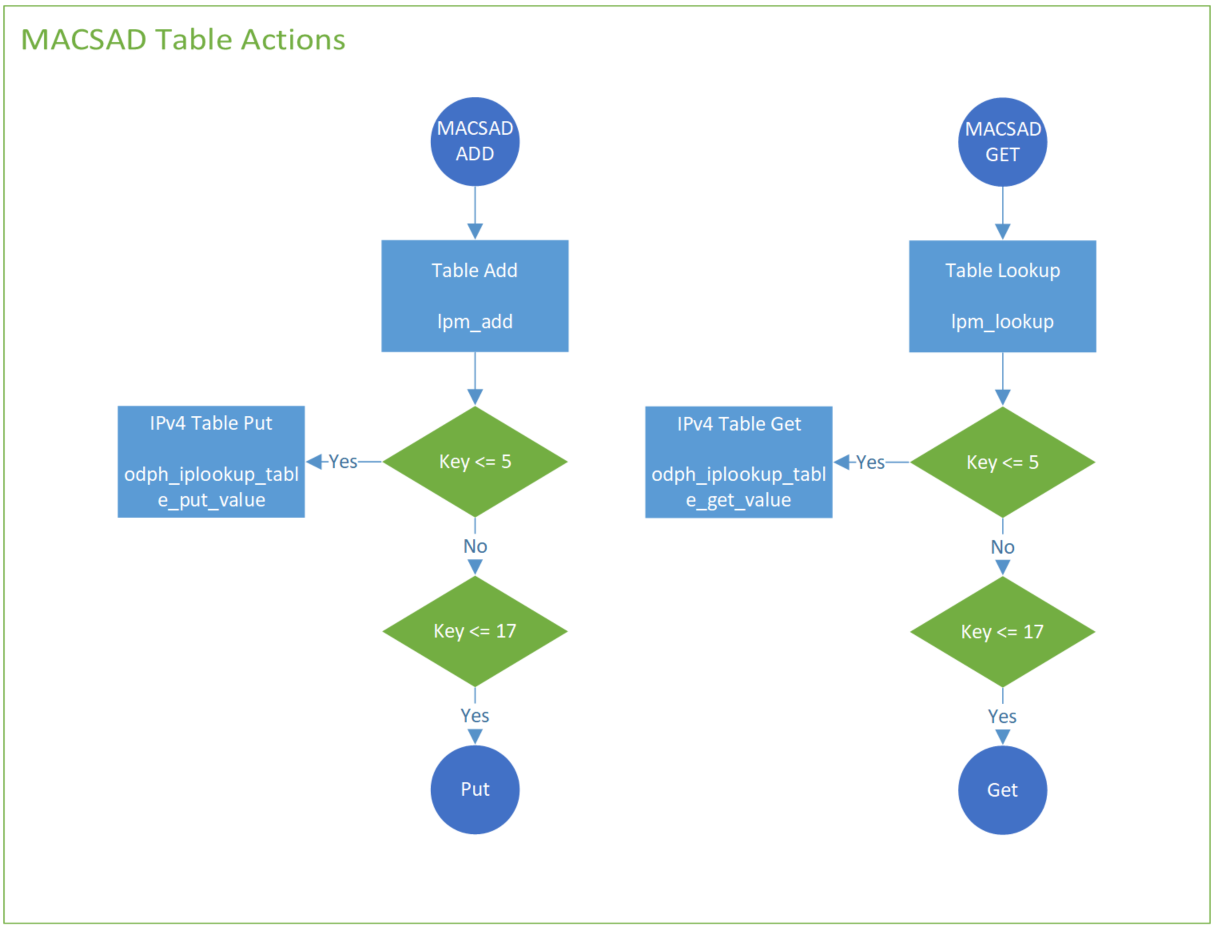


Figure 21 – ODP Table Actios

4 Packet Generator Tool: BB-Gen

With P4 gaining traction to define datapath pipelines along auto-generated control plane APIs, the protocol-independence and increased flexibility add non-trivial hazards when it comes to functional and in-depth performance evaluation. P4-dependent workload traces are needed along automated methods to populate the tables of the datapath under test accordingly. Without proper tools, manual efforts are required for tedious tasks such as creating appropriate PCAP traces, defining the distribution of field values, and inserting entries in the pipeline tables. To this end, we developed BB-Gen (RODRIGUEZ *et al.*, 2018; CESEN *et al.*, 2018), a packet crafter and table generator tool that given a P4 application and a corresponding user configuration results in packet and table traces to carry automated performance evaluation tasks. We evaluated BB-Gen with P4 applications of increasing complexity (from L2 to VXLAN-based Data Center Gateway), using two different multi-architecture backend compilers (MACSAD, T4P4S) and different targets.

With increasing number of services over the Internet like email, web, video streaming and so forth, the demand for bandwidth is increasing exponentially. Along with it, the necessity to evaluate and test network capabilities become prevalent. While networks are becoming more (re-configurable), network testing tools are becoming equally complex adapting to the need of the hour. The network testing and benchmarking tools depend on network workload generation to simulate the network traffic for testing purposes. This trivial task has been the foundation for several research activities like (BOTTA *et al.*, 2012) focusing towards performance, scalability, and reliability of networks and network devices.

Traffic generator tools are an essential part of network testing with features ranging from supporting list of protocols, analyzing network traffic or measuring throughput to calculating latency of packets. In theirs strive to achieve feature completeness, the tools are getting more complex each time, and making it hard to port, manage, and use. To address this, we propose BB-Gen which is a python based tool, with a primary focus on simplicity, excelling in the creation of network packet traces.

BB-Gen is a simple CLI based packet crafter written in Python over Scapy library. It can natively craft packets for different standard and custom protocols. It aims to create PCAP files to be used with a wide set of Traffic Generators (e.g., pktgen-dpdk (OLSSON, 2005), NFPA (CSIKOR *et al.*, 2015a; CSIKOR *et al.*, 2015b), TCPDUMP (TCPDUMP, 2010)) helping network developers to validate the network and execute performance tests over the targets.

Though BB-gen is primarily used to create PCAP trace files, it differs itself by generating the table trace files for the PCAPs which are necessary to fill the table flow configuration of the target device for the network testing. Table traces contains the main information of the generated packets (e.g. source/destination IP/MAC address). BB-Gen allows to create traces files with same/random IP/MAC/L4Port details showing its control over the header fields like source and destination MAC addresses, IP addresses, TCP or UDP ports while creating packets. It also allows the user to create a complete set of PCAPs for performance test by specifying a single flag in the command line. Under this performance setting, PCAPs generated comprises of all the standard packet sizes (64, 128, 256, 512, 1024, 1280, 1518) (BRADNER; MCQUAID, 1999) and also features simple (best-case) and complex (worst-case) scenarios by using typical/random distribution sets of header fields respectively. A single command can generate both PCAP and table trace files. The command line arguments which are reckoned and self explained go in agreement with the easy use of BB-Gen. A custom protocol support to BB-Gen can be easily added by first adding the support to Scapy similar to the Contrib[1] and then extending BB-Gen protocol list with minimal code changes.

4.1 Architecture

Figure 22 shows the principal components of the architecture of BB-Gen Packet Crafter.

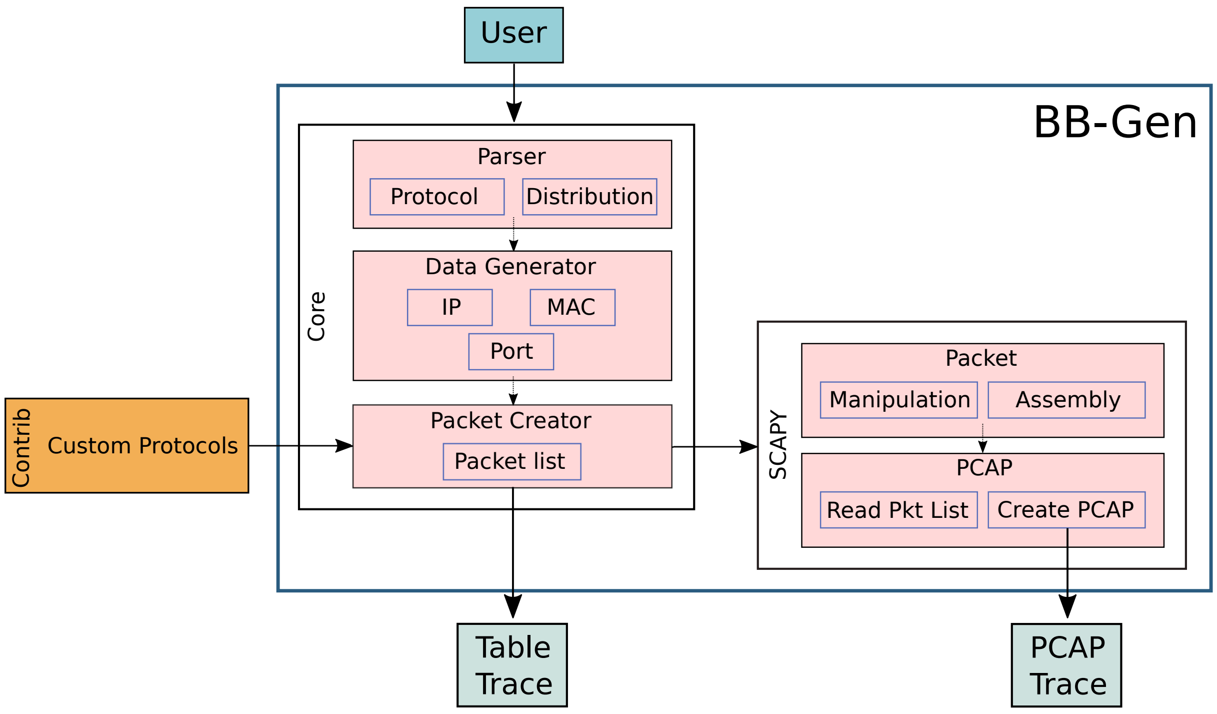


Figure 22 – BB-Gen Architecture

[1] https://github.com/secdev/scapy/tree/master/scapy/contrib

- ***User:*** The user introduces required parameters such as distribution, protocols, numbers of entries, use case[2], packet sizes, necessary to create the trace files. [3]

- ***Core:*** Being the principal part of BB-Gen, it receives and process information from the *User*, and generates the packet details to be included in the trace's files. It comprises of three sub-modules i.e., Parser, Data Generator, and Packet Generator as explained below (Fig. 23).

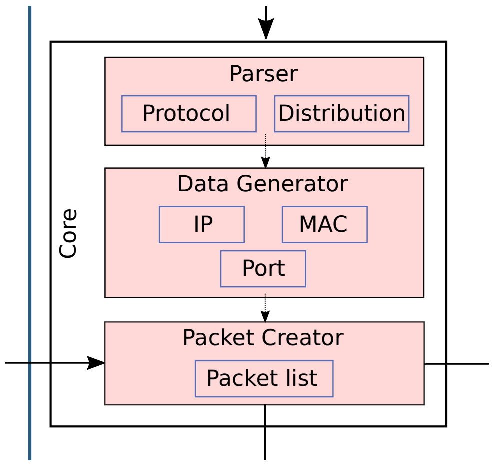


Figure 23 – BB-Gen Core module

  – ***Parser:*** it is in charge of selecting the protocols to be used as well as the distribution, using the information introduced by the use or the default values in case of missing information (Fig. 24).

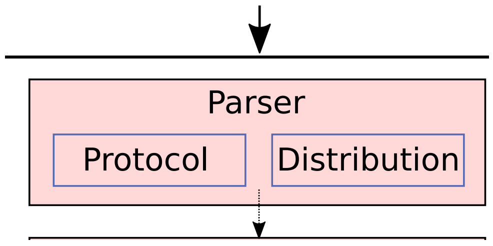


Figure 24 – BB-Gen Parser module

  – ***Data Generator:*** using the protocols and the distribution details from *Parser*, it generates the list of source and destination IP, MAC and Ports (Fig. 25).

  – ***Packet Creator:*** with the information set at *Parser* and the list of IP, MAC and Ports generated at *Data Generator*, the Packet Creator is going to create the list of packets with all the defined fields. With the list of packets prepared, the table trace file is going to be created using the informations about packet

[2] Supported use case: MACSAD

[3] More information on required parameters to generate the traces are described in BB-Gen GitHub Wiki page, <https://github.com/intrig-unicamp/BB-Gen>

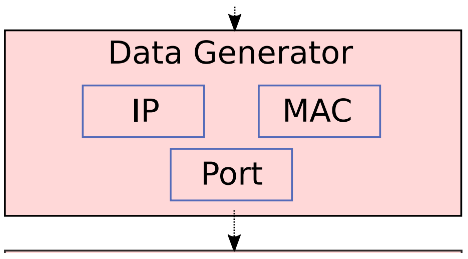


Figure 25 – BB-Gen Data Generator module

contents. And finally, the list of packets is sent to the Scapy block to generate the final PCAP trace (Fig. 26).

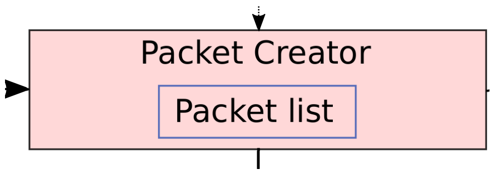


Figure 26 – BB-Gen Packet Creator module

- ***Scapy:*** is composed of the *Packet* and *PCAP* sub-blocks. The *Packet* is going to assemble the packets included in the list of packets with the correct protocol format. The *PCAP* will read the assembled packets and generate the PCAP file completing the BB-Gen process (Fig. 27).

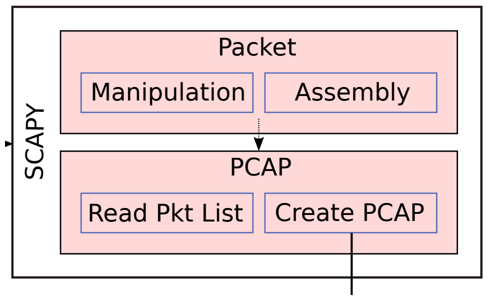


Figure 27 – BB-Gen Scapy module

- ***Output Files:*** BB-Gen generates two output files, the Table trace and the PCAP trace:
  - ***Table Trace:*** Generated by the Packet Creator module, it is a plain text containing matching entries derived from the PCAP packet trace and Packet List. It can be customized to the specific use-case and used to populate entries to the dataplane tables.
  - ***PCAP Trace:*** Generated by the Scapy module with all the information and protocols configured by the user. It can be used with a wide set of benchmarking tools for the evaluation test.

4.2 Main features

The principal features and capabilities of BB-Gen are summarized as below:

- Designed for simplicity, BB-Gen delivers an intuitive CLI based interface. By specifying only a few flags, the user can create a set of traces files.
- A P4 code can be used as an Input. BB-Gen identifies the protocols and auto-generate the traffic traces.
- Very useful for best-case and worst-case testing. It allows to specify a simple/random distribution of header fields sufficient to address the most complex test cases.
- Being a python based tool, it is easy to build, use and extend to support additional protocols and new features.
- Easily create multiple PCAPs in a single step. The user can define the number of flows, packet sizes, for each PCAP.
- Generates table trace files along with every set of PCAPs utilizing the informations from the PCAP files such as list of IP addresses, MAC address, Port numbers and also the packet encapsulation data for protocols like VXLAN and GRE. Trace file generation is seamless and does not require any additional user input.
- For scalability testing purposes, it can generate traces with more than 1 million unique packet details.
- Supports a list of common standard protocols:
  - Ethernet.
  - IPv4, IPv6.
  - TCP, UDP.
  - Protocol Encapsulations such as GRE and VXLAN.
- Useful for performance tests as it can automatically create packets of different sizes according to the RFC 2544 (BRADNER; MCQUAID, 1999) (64, 128, 256, 512, 1024, 1280, 1518 Bytes) by setting a single performance flag in CLI.
- User defined custom packet sizes are also accepted at the CLI, just being limited by the defined minimum protocol size.
- Accepts user defined payload information. For this scenario, minimum packet size is maintained to be 64 Bytes by padding with random strings if necessary. In case payload saturates the 64 Bytes, the packet size is determined by the aforementioned payload.

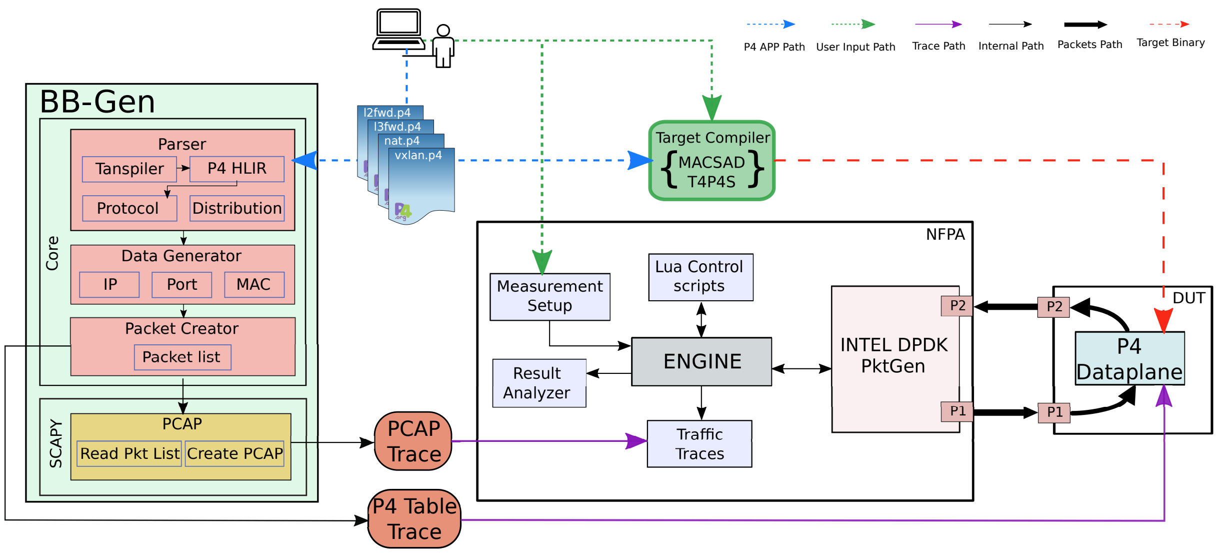


Figure 28 – BB-Gen integration with NFPA and MACSAD & T4P4S

- The generated PCAP trace files are accepted as *inputs* for different network benchmarking and performance tools.
- It is a cross-platform tool with support for Windows, Linux, BSD and Mac OS X platforms.
- It is an open source project following BSD 3-Clause License.

4.3 Use Case

In order to demonstrate the usability of BB-Gen, we present a use case featuring a programmable dataplane (MACSAD & T4P4S) and a network performance evaluation tool (NFPA) which accepts a set of PCAP trace files as input for each 'determined setup'/'specified configuration' experiment run.

4.3.1 Evaluation

While using the NFPA benchmarking tool for performance evaluation of MACSAD & T4P4S, BB-Gen generates the necessary PCAP and table trace files for worst-case scenarios with random header field values (MAC and IP addresses, Port numbers.) from the P4 file set as an input. Multiple sets of PCAPs are created for different packet sizes according to the RFC 2544, and also with a different number of packet flows (100 to 1 million unique flows). This is repeated for each use-cases supported by MACSAD & T4P4S such as L2-Fwd, L3-Fwd with IPv4 and IPv6, GRE, and VXLAN.

For the use case evaluation, the NFPA standalone node is connected to the DUT (MACSAD or T4P4S) as presented in Figure 28 (A detailed use case is presented in Chapter 5). The user defines his P4 program and configures the benchmarking tool accordingly (in our case, NFPA only requires the high-level configuration details, e.g., packet size). Then, the P4 program is fed into BB-gen. Users can define target PCAPs with different packet sizes (i.e., from 64 to 1500 Bytes) from best-case (i.e., fixed, single header fields) to worstcase (random, unique field values) workloads. While the packets are generated, the DUT is compiled by our Target Compiler module supporting multi-architecture P4 compilers, such as MACSAD and T4P4S.

Once the DUT is running, BB-gen uses the available APIs[4] to carefully populate the flow tables according to the P4 Table Trace containing matching entries derived from the PCAP packet trace and Packet List. At the same time, the BB-gen loads the generated PCAP file into NFPA, which takes care of the practical measurement conforming the standards.

4.4 Summary

In this Chapter, we presented BB-Gen a packet generator tool that can generate a suitable trace file used for performance evaluation with a simple process. BB-Gen can also create multiple sets of trace files with different packet sizes and different flow distribution simultaneously along with the corresponding table trace file for each PCAPs and pipelines. The integration with an extensive set of benchmarking tools reinforces the ease of use of the tool and benefits for the community with the evaluations.

We showed how BB-Gen, NFPA and "MACSAD & T4P4S" trio can work conjointly to exemplify how a P4 program suffices to define the datapath pipeline, create match + action table traces to control the P4 DUT, and generate a trace file for the evaluation, covering different complexities and requirements of the project.

[4] As today, APIs are target-specific but P4Runtime (PRT) API support is underway.

5 Experimental Evaluation

In this chapter, we evaluate the performance of the two LPM use cases using three different packets I/O engines (DPDK, Netmap, and Socket_mmap). For each combination, we explore the scalability for different workloads (packet traces, table entries) and configuration options (e.g., CPU cores) using NFPA as a benchmarking tool. To generate the traces we use our packet crafter tool BB-Gen[1] that will provide the necessary PCAP files to be used with NFPA. The pipeline implementation[2] and other information for reproducibility purposes including the P4 programs[3] used by MACSAD and the traffic generator tool (BB-Gen) are publicly available.

5.1 Testbed and Methodology

Our testbed (Fig. 29) contains two Lenovo ThinkServer RD640 servers with Intel Xeon E5-2620v2, 6 Cores, Hyper-Threading disabled, running at 2.1GHz, 8*8GB DDR3, a dual-port Intel X540-AT2 NIC (10G), and run with Ubuntu Linux 16.04 LTS (kernel 4.4). The Tester server (module 5 in Fig. 29) runs NFPA with DPDK v17.08 and PktGen v3.4.5, and it is connected to the DUT (BRADNER; MCQUAID, 1999) (module 4). The DUT supports multiple packets I/Os to illustrate the ability to accommodate different platform features, such as DPDK v17.08, ODP v1.16.0.0, Netmap v11.2, and the basic Linux Socket_mmap provided by the Linux kernel. The P4 Code (module 1) with the Ipv4 or IPv6 implementations is the first input of the test. The code will be used by BB-Gen (module 2) to auto-generate the PCAP and the table Trace, and by the compiler (module 3) to generate the MacS using the ODP APIs. The controller parses the table Traces file and populated the information into the corresponding table. The PCAP files are used by the NFPA to generate the traffic for the performance evaluation. The MacS is configured to forward packets received from one port to the other and eventually back towards NFPA, which in turn analyzes the packet throughput concerning pps and bps (module 6).

For both L3-IPv4 and L3-IPv6, different number of cores (1, 2, 4, and 6) are allocated to the DUT, distinctive workloads are configured by setting different number of IP prefixes (100, 1K, 10K, 100K, 1M) in the lookup table and a matching number of L3 flows used in the synthetic traces.

[1] <https://github.com/intrig-unicamp/BB-Gen>
[2] <https://github.com/intrig-unicamp/macsad>
[3] <https://github.com/intrig-unicamp/macsad-usecases>

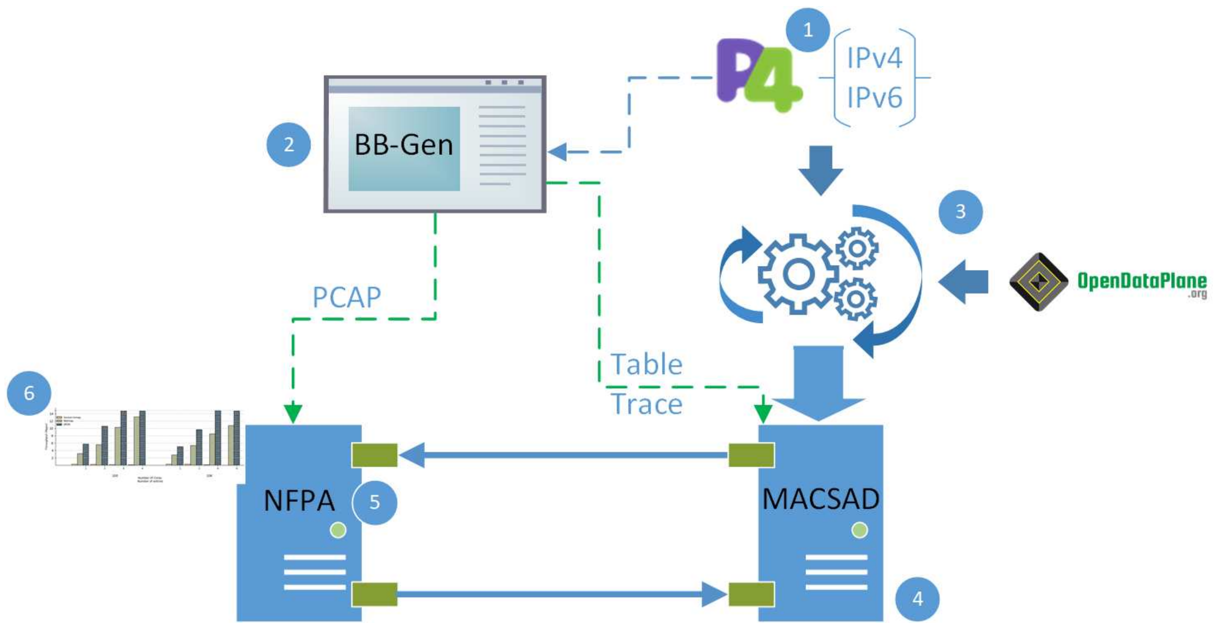


Figure 29 – Testbed

5.2 Performance Analysis

5.2.1 L3-IPv4

Figure 30 shows the performance of L3-IPv4 for different FIB sizes and packet I/O drivers, i.e., DPDK, Netmap, and Socket_mmap. The red horizontal lines refer the line rate for different packet sizes. (i.e., 8.44 Mpps for 128 bytes and 4.52 Mpps for 256 bytes). The results for L3-IPv4 are grouped into three sectors indicating different packet sizes (i.e., 64, 128, 256). Each sector is further divided into five different points marking the complexity of the pipeline, i.e., the size of the FIB (e.g., 100, 1K). It can be observed that MacS, with DPDK, reaches the 256 bytes' packet size line rate regardless of the FIB table size. The performance of Netmap is comparatively lower, but it reaches line rate with 512B packets. Also, it is interesting to note that, the measured results for 1M FIB entries are better than for 100K FIB entries. From the results, it is clear that the Linux Socket_mmap driver never saturates the 10G interfaces even with the largest packets (1518 bytes) due to the highly increased number of system calls, fundamental kernel scheduling, and costly context switching imposed by the Linux kernel itself.

In a configuration using 4 cores (Figure 31) the results of the different packet I/O drivers where grouped depending on the FIB size and the packet size. It is important to notice that all the results independent of the driver are starting from '0' in the 'y' axis. It is clear that the performance increases notably compared to the 1 core setup. With DPDK the line rate is reached even with small packets of 64 bytes and with all the FIB sizes. Netpmap also increases the performance and can saturate the link with packets of 256 bytes. It is interesting to note that the performance results with 64 bytes'

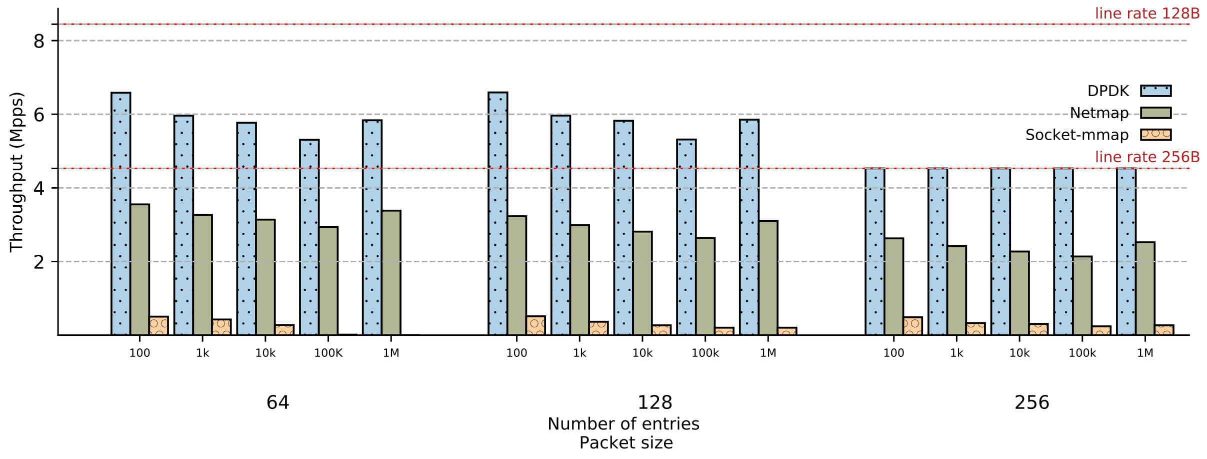


Figure 30 – IPv4 forwarding performance for different I/O drivers (1 CPU core)

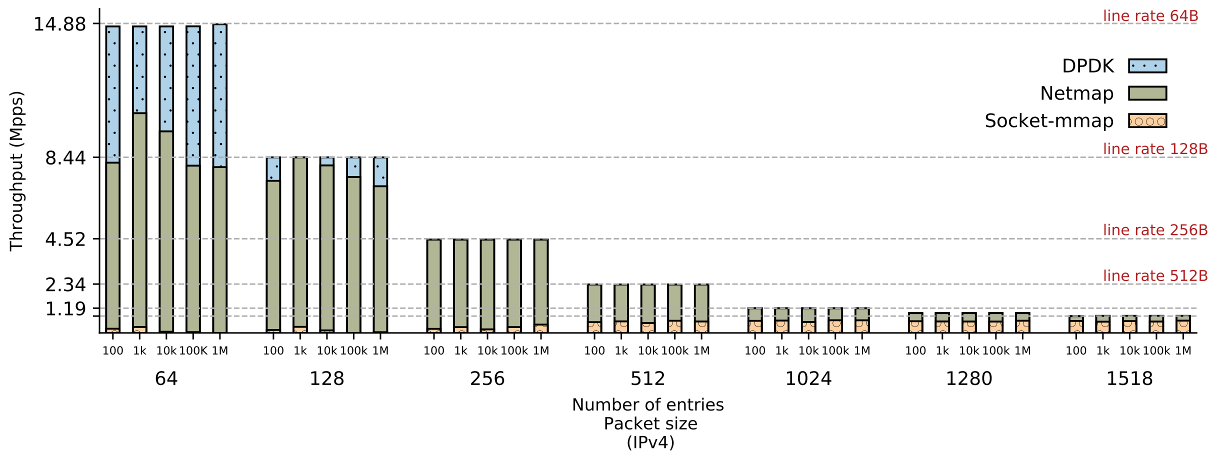


Figure 31 – IPv4 forwarding performance for different I/O drivers (4 CPU core)

packets and 1K entries are better than the 100 entries results. This behavior can be caused by a sub-optimal synchronization and CPU utilization of Netmap, it uses NIC interrupts and standard kernel synchronization mechanisms to block on empty or full NIC queues (LETTIERI *et al.*, 2017). With Socket_mmap driver are archived slightly superior results if we compare with the performance of the 1 core evaluation using the same driver.

Figure 32 presents the results for different core configurations (i.e., from 1 to 6), and a 64 bytes' packet size. The results include a FIB size of 100 and 10K. Our results demonstrated that the performance increase with the addition of cores, this analysis applies for the DPDK and Netmap drives, with both configurations and independent of the FIB size it is clear that throughput raise. In case of DPDK, the line rate is reached with 4 cores, as it was demonstrated before, with Netmap the link is saturated only with packets starting from 256 bytes. A similar pattern of the previous results was obtained with Socket_mmap, confirming that the limitations of the Linux kernel affect the performance using this driver.

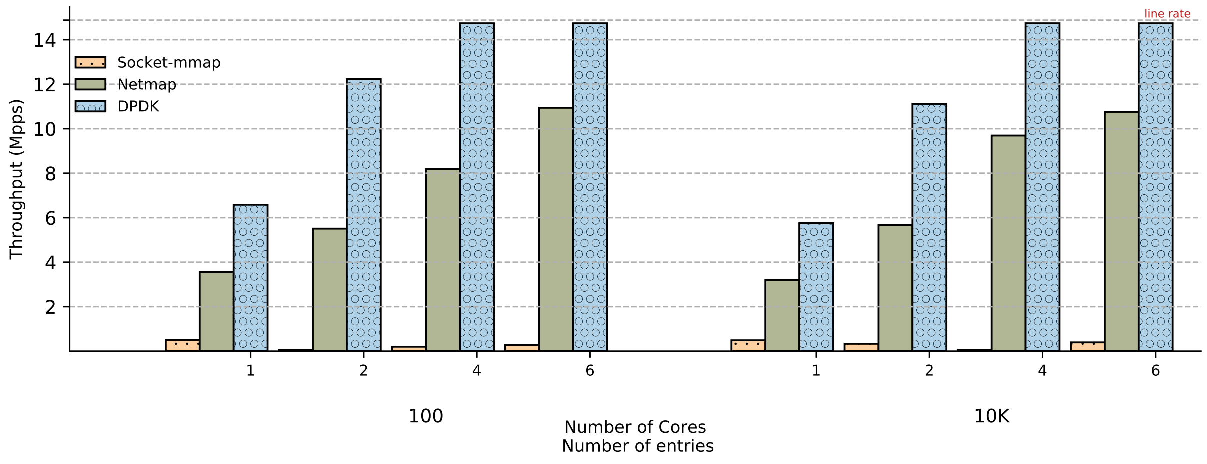


Figure 32 – IPv4 different cores performance (64 bytes packets)

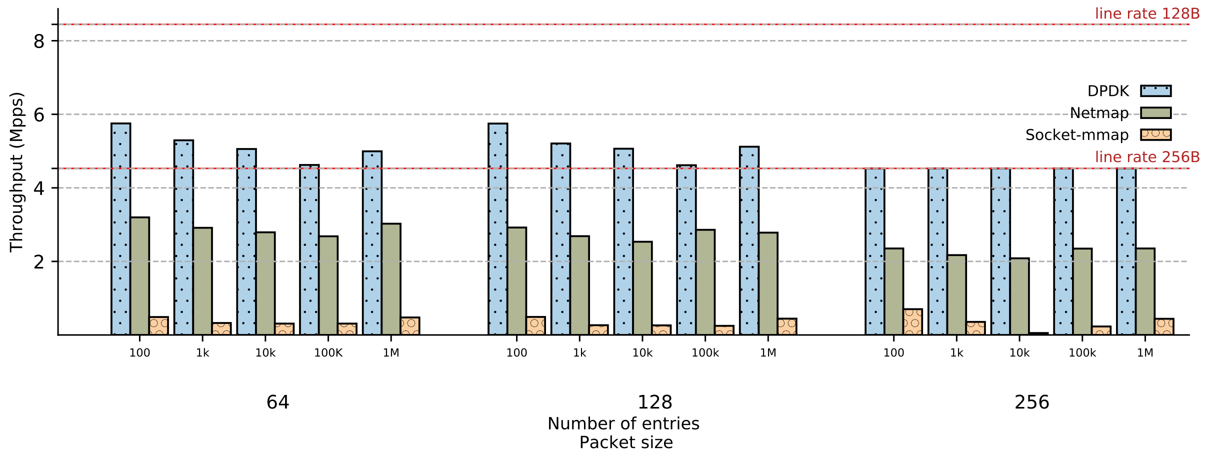


Figure 33 – IPv6 forwarding performance for different I/O drivers (1 CPU core)

5.2.2 L3-IPv6

Results for the L3-IPv6 use case with 1 CPU core are shown in Figure 33. The performance results lead to a similar conclusion to L3-IPv4 where DPDK reaches line rate (red line) with 256 bytes' packets for all FIB sizes. There are some performance differences in case of the Netmap driver, a slight drop as the number of FIB entries grows. However, when comparing our results to the L3-IPv4, we must point out that the peculiarity with 100K and 1M number of entries observed before also applies for L3-IPv6.

Comparing the IPv6 results with IPv4 (Figure 34, it is important to highlight the fact that the performance in the case of IPv4 is slightly superior with different packets sizes (i.e., 64 and 128 bytes). Additionally, it is remarkable that in the case of IPv6 the Key size is 128 bits increasing the complexity of the LPM tree.

The IPv6 performance results with a four cores configuration are presented in Figure 35. It is important to highlight, with Netmap when the FIB size reaches 1M the line rate is not achieved even with the largest packet size, the limitations of Netmap can

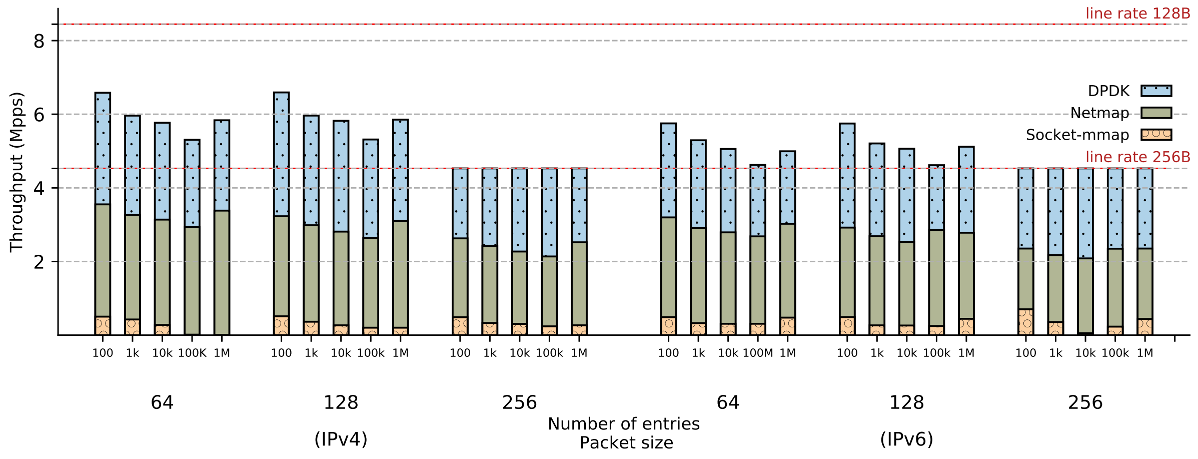


Figure 34 – IPv4/IPv6 forwarding performance for different I/O drivers (1 CPU core).

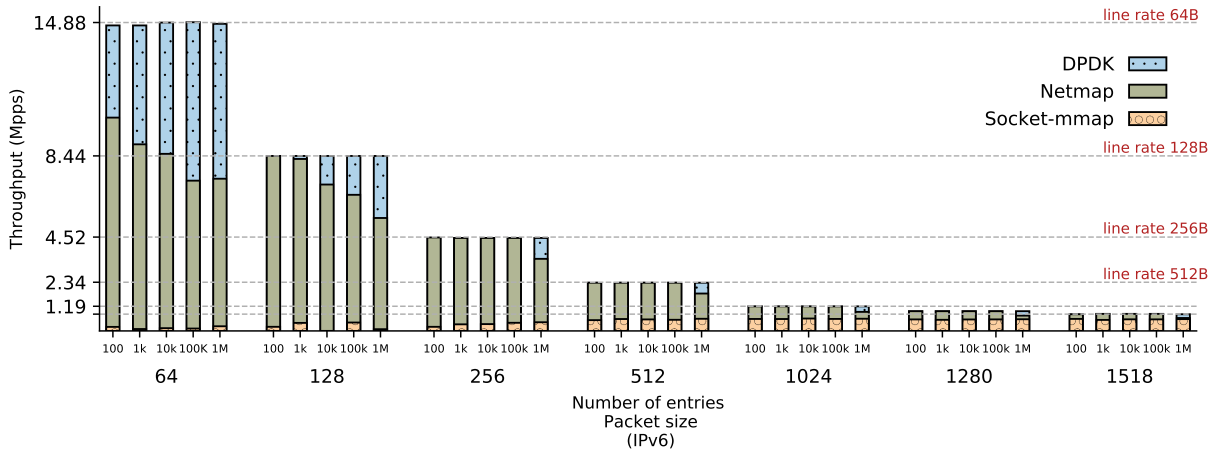


Figure 35 – IPv6 forwarding performance for different I/O drivers (4 CPU core)

cause this behavior.

Figure 36 shows a throughput comparison as the number of cores increases from 1 to 6. The results with different cores are consistent with what has been discussed before. When the number of CPU cores increases, MACSAD can process more packets resulting in higher throughput.

It is notable (red line in the Figure 37) that as the number of FIB entries increases, the throughput reduces slightly. Moreover, when the table key size increase from 32 bytes (IPv4) to 128 bytes (IPv6) the performance also decreases. This is a significant finding in the understanding of how the complexity of the number of FIB entries and key sizes affect the throughput.

Figure 38 presents the performance results of the IPv6 lookup by setting different values to the prefix length. It is interesting to note that, when the key value (prefix) increases, the performance reduces. With a key of 16 and 32, the line rate is easily reached. When we increase the prefix to 64, the throughput decreases to 12.9 Mpps, and while

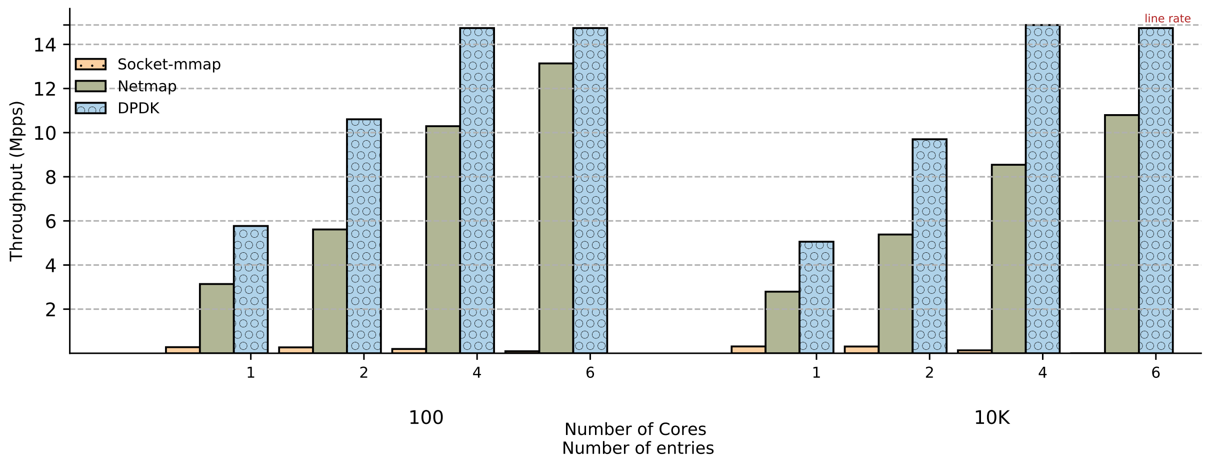


Figure 36 – IPv6 different cores performance (64 bytes packets)

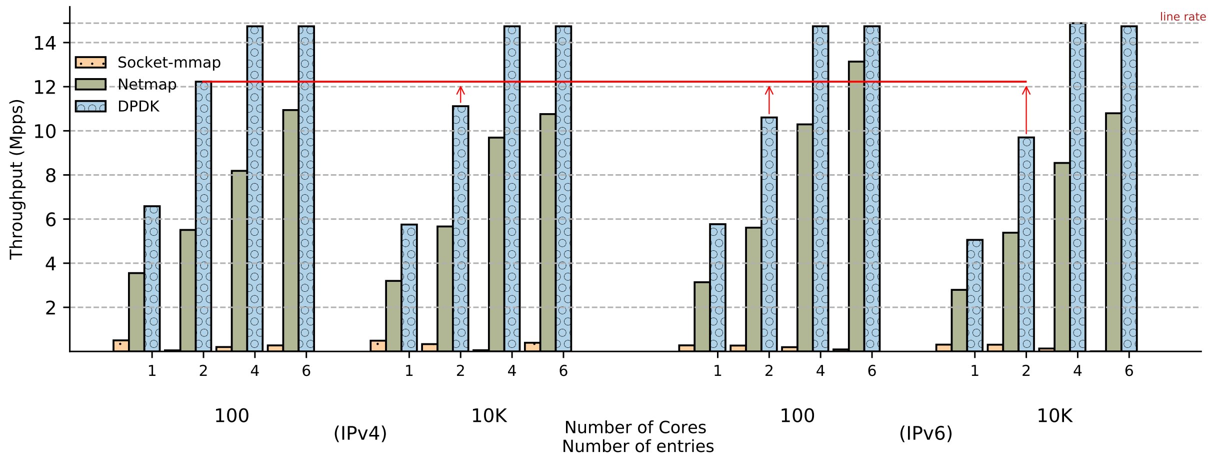


Figure 37 – IPv4/IPv6 different cores performance (64 bytes packets)

increases the prefix the performance also reduces. This implies that the throughput is associated with the size of the key. In ODP library when we increase the value of the prefix, the algorithm has to create more leaves, and when we perform a lookup, it is necessary to search in all the levels adding processing and memory consumption.

Comparing MACSAD IPv4 and IPv6 implementation with T4P4s, we can observe in Figure 39 that the performance of the IPv4 use case is slightly superior in MACSAD. This can be caused by an optimized algorithm to perform the LPM lookup. While comparing the IPv4 and IPv6 implementation with the l2fwd use case, we can observe that the L2 forwarding has the worst performance result. From these results, it is clear that exact lookup used by the L2 use case is not optimized what causes a reduction in the performance.

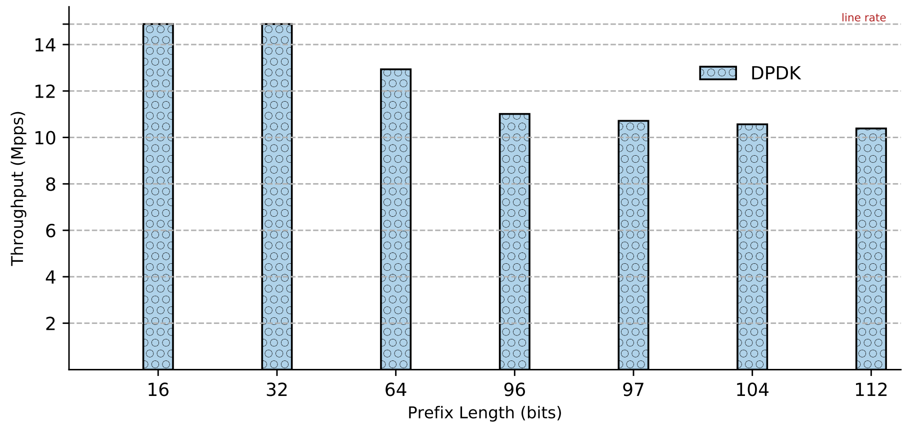


Figure 38 – MACSAD IPv6 prefix length comparison (64 bytes packets, 100 entries)

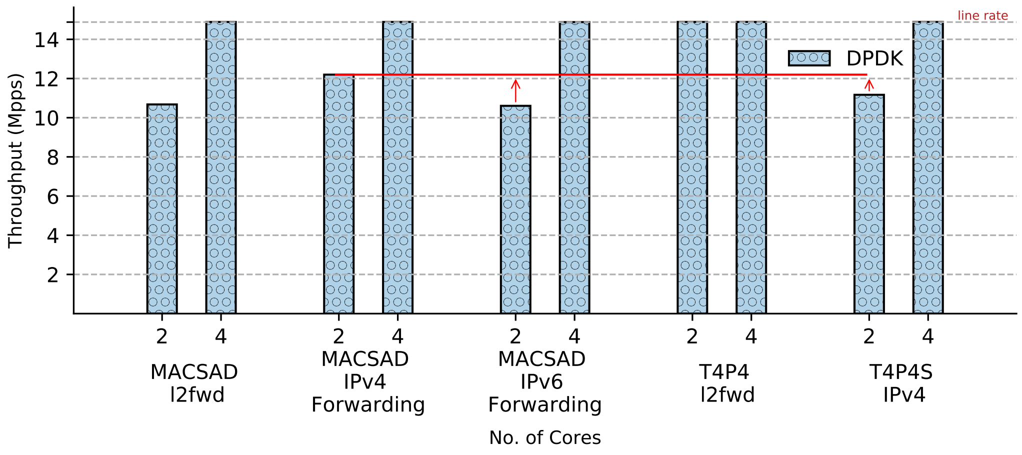


Figure 39 – MACSAD vs T4P4S use cases comparison (64 bytes packets, 100 entries)

5.2.3 CPU Cycles

Performance counters are special hardware registers that are available on most of the CPUs and count the number of some hardware events (i.e., instructions executed, cache-misses suffered, branches mispredicted (Without slowing down the kernel or applications). The perf command (MELO, 2010) is a powerful tool that can instrument CPU performance counters, tracepoints, kprobes, and uprobes (dynamic tracing). Other similar tools to analyze the CPU performance are: the Berkeley Packet Filter (BPF) (MCCANNE; JACOBSON, 1993), a register-based filter evaluator for filtering network packets (best known for its use in tcpdump), and the Extended BPF (eBPF) variant that analyzes all over the kernel, including maps and used for aggregating statistics of events.

We evaluate the CPU Cycles using Perf and with the same topology of the performance analysis (Fig. 29). For the evaluation, we use a configuration of 1 CPU core and 100 entries, the prefix size of 24, and DPDK as the packet I/O due to the best results during the performance tests. We compare MACSAD and T4P4S; both projects use a similar structure with the difference that T4P4S uses a DPDK backend and libraries,

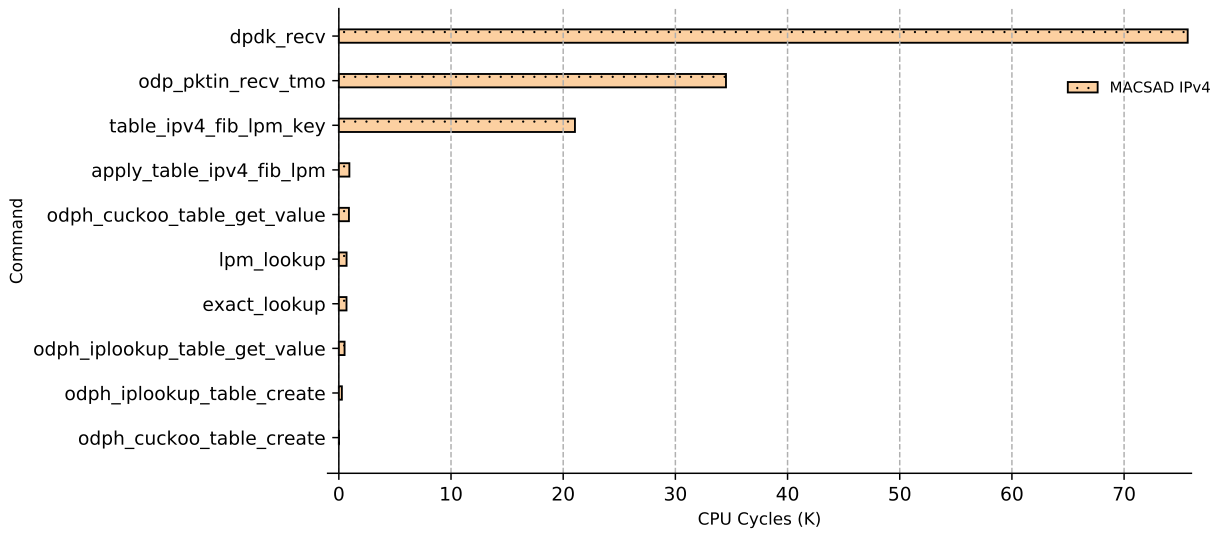


Figure 40 – MACSAD IPv4 CPU Cycles (324k total cycles, 1 Core, 100 entries)

including the LPM functions.

In Figure 40 it can be seen the results of the IPv4 test, are considered for the graph the results related to the forwarding process. During the test were captured 324k CPU cycles in total. It is important to remark that the commands that process the receiving packets (i.e., *dpdk_recv*, *odp_pktin_recv_tmo*, *ixgbe_recv_pkts_vec*) consume the largest number of cycles (155k CPU cycles). The *table_ipv4_fib_lpm_key* command start the LPM processing using 21k CPU cycles. The MACSAD LPM function of lookup *lpm_lookup* takes 689 CPU cycles samples, inside this command the ODP get function *odph_iplookup_table_get_value* takes 511 CPU cycles. The ODP table create *odph_iplookup_table_create* and table add use few CPU cycles (less than 200). Comparing the LPM commands with the cuckoo commands, it is clear that the LPM functions have a high impact in the CPU cycles consumed.

For the IPv6 case (Figure 41), the results lead to a similar conclusion of the IPv4 results, where the commands that process receiving packets consume more cycles, from the 261k captured more than 100k correspond to the receiving process. In the case of IPv6, the LPM functions consume more CPU cycles; the LPM lookup reaches the 2.6k CPU cycles. The results confirm that the size of the key affects the processing of the packets adding more cycles. It leads to good results, even if the processing in IPv6 is higher, confirming the results from our evaluation of different cores and use cases IPv4/IPv6 where we saw a reduction on the performance when the size of the key increases.

When comparing our results with T4P4S (Figure 42), it worth mentioning that MACSAD uses less CPU cycles. In the case of our LPM lookup, MACSAD IPv4 takes 689 CPU cycles, MACSAD IPv6 2.6k, and T4P4S with DPDK LPM lookup takes 12.9k. With the evaluation, it is possible to see significantly better results for MACSAD using the our implemented ODP library for the LPM process than T4P4S using DPDK library

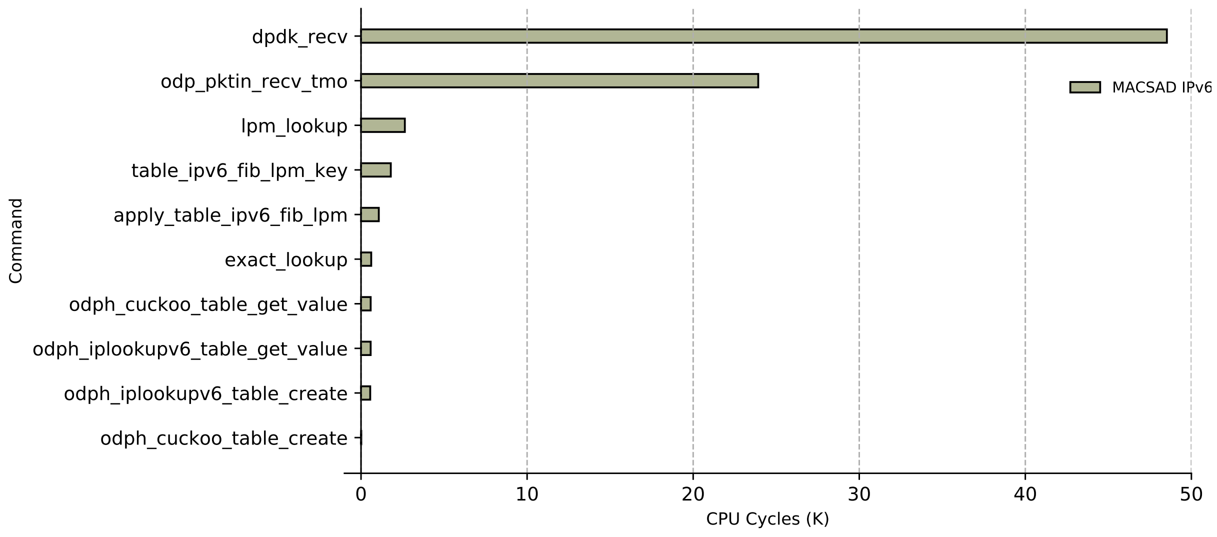


Figure 41 – MACSAD IPv6 CPU Cycles (261K total cycles, 1 Core, 100 entries)

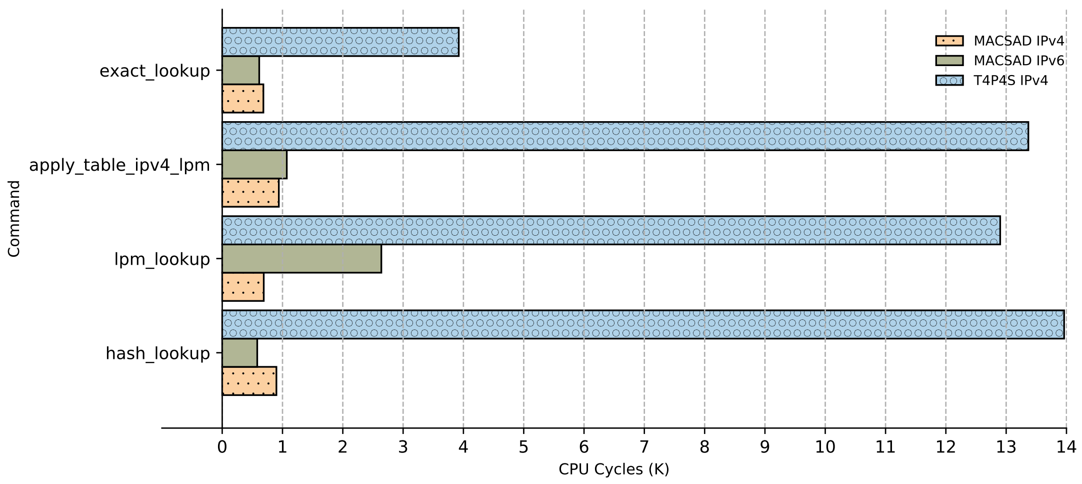


Figure 42 – MACSAD IPv4/IPv6 vs T4P4S - CPU Cycles - LPM and exact lookup commands (1 Core, 100 entries)

for the LPM process. One difference between our ODP LPM library and DPDK LPM library can be attributed to an optimized algorithm to store the LPM entries in the table, which can reduce the number of cycles consumed by the process. This is consistent with what has been found in our performance evaluation (Figure 39).

5.3 Summary

The performance evaluation is a crucial part of the development of a system. We evaluated our LPM implementation with MACSAD in different scenarios and configurations, including different sets of packet I/O drivers, cores, numbers of entries, packet sizes, and much more. We compared our results with other similar projects like T4P4S, analyzing the performance and the CPU Cycles consumed by the projects. We presented an analysis of how the prefix length can impact the performance and how the size of the prefix tree (number of levels) also impacts on the results.

6 Conclusions and Future Work

The focus of this dissertation is the design, implementation, and evaluation of IPv4/IPv6 Longest Prefix Match support in Multi-Architecture Programmable Dataplanes to address the limitation of IPv4 lookup support, lack of IPv6 support in ODP, and understanding the performance capabilities. To this end, we have: (i) implemented and evaluated the IPv4 and IPv6 LPM support in MACSAD, (ii) developed a new ODP Helper library for the IPv6 lookup mechanism based on the current IPv4 solution, (iii) carried performance and experimental evaluation of scalability, (iv) developed a P4 CLI-based packet crafter to generate packet flows formatted as PCAP files, and (v) released all artifacts as open source.

This dissertation describes the IPv4 LPM support implementation and a detailed IPv6 implementation including the functions and processes, including the Controller and the P4 Code design. We compared the complexity of three different P4 use cases, i.e., L2-FWD, L3-FWD with IPv4 and IPv6, remarking the different sizes of headers, tables, and lookup types that impact performance evaluation. We learned that the performance is affected by the lookup type and the size of the key, which are key characteristics of the use case pipeline complexity (e.g., number of tables, key size, table size).

We evaluated the performance of the LPM use cases using three different packet I/O engines (DPDK, Netmap, Socket_mmap). We explored the scalability for different workloads (packet traces, table entries) and configuration options (e.g., CPU cores), using NFPA to transmit the PCAPs created by BB-Gen. Comparing the performance of the different packet I/O drivers, we confirmed that Socket-mmap is slower as it is the Linux default driver without any fastpath advantages from Netmap or DPDK. Netmap performance is lower than DPDK on ODP system because in the case of the Netmap packet the copy operation becomes costlier as in the case of the DPDK support, the ODP implement a zero-copy feature. With our findings, we can understand that with the use of the Netmap driver we have a sub-optimal CPU synchronization and CPU utilization, the reason is that Netmap uses NIC interrupts and standard kernel synchronization mechanisms to block on empty or full NIC queues. In conclusion, DPDK seems to have a better performance, due to its packet-processing optimized design and the circumvention of the kernel space.

Comparing the IPv6 results with IPv4, the performance in the case of IPv4 is slightly superior for different packets sizes (i.e., 64 and 128 bytes). This conclusion follows from the fact that the Key size in IPv6 is 128 bits increases the complexity of the LPM tree and affects the performance. Broadly translating, our findings in the performance tests indicate that the CPU cycles consumed by or ODP LPM implementation are less than the DPDK implementation even in the case of IPv6, this is confirmed by the performance results where MACSAD presents a better result.

These experiments add to a growing corpus of research showing that P4 and ODP can work in conjunction to create a good base for a programmable dataplane device. P4 is gaining interest by the community, and it is in constant development and maintenance, as well as ODP that continues emerging as a good set of APIs for the dataplane backend.

The BB-Gen packet generator tool allows creating the PCAP and table trace files for the performance evaluation. BB-Gen can also create multiple sets of trace files with different packet sizes and different flow distribution simultaneously along with the corresponding table trace file for each PCAP and pipeline. BB-Gen effectively complements other tools of the P4 developers gadgets, such as `p4pktgen` that validates a P4 program by generating test input traffic and fills the tables, `p4app` that performs functional tests using BMV2 simple_switch in Mininet environments for P4 programs, and `P4benchmark` that tests the target compilers generating different P4 applications varying the complexity.

We tested the applicability of BB-Gen in real projects i.e., MACSAD and T4P4S, by creating the traces and PCAPs from a P4 file for different use cases with varying complexities, suggesting that other developers could benefit from our tool in their evaluation workflows, as well as the growing community striving for research reproducibility, e.g., by re-using traces, toolchains, and evaluation methodologies. With BB-Gen, NFPA and "MACSAD & T4P4S" we demonstrated that they can work conjointly to illustrate the datapath pipeline defined by a P4 code, create match+action table traces to control the P4 tables in the DUT, and generate a suitable PCAP file used for performance evaluation of different setups.

Altogether, this work accomplished a series of open source contributions: (1) the IPv6 lookup library developed will be suggested for adoption by ODP project, (2) BB-Gen, (3) P4 pipelines for IPv4 and IPv6, and (4) various trace files for the NFPA repository.

Future Work

To conclude, we list some future tasks and open questions worth further investigation.

Related to BB-Gen:

- Integration with `P4Runtime` [1] to control the DUT, leveraging the possibility to configure the P4 table entries by adding, updating, deleting, along with others.

[1] <https://p4.org/p4-runtime/>

- BB-Gen will allow to rely on Scapy's extensibility features to create packets with arbitrary sequence of headers that might not conform to standards, but an unorthodox P4 program may require it.

- Although Scapy is a powerful Python library to easily generate any type of packet, we have experienced that generating hundreds of thousands of packets can take several minutes, however since the performance evaluations are usually done offline (i.e., before an actual deployment), the measurements do not require real-time traffic generation. Therefore, we also take into consideration of Hexdump generation of packets proven to be more effective[2] and contribute to materialize other open-source projects into BB-gen.

Related to the LPM implementation:

- A deep analysis in how the performance is affected by the variation of the prefix length. In this work, we covered a brief overview of the impact of the prefix length in the performance. This provides a good starting point for discussion and further research, analyzing real distributions and how distribution can be optimized in a forwarding device.

- We defined the prefix tree levels following the ODP IPV4 specification. It will be important that future research investigate if there is an optimal level distribution for IPv4 and IPv6, and how this distribution can affect the performance of the device.

- With the evaluated performance properties, e.g., packet loss, latency, and CPU cycles. we demonstrated that the ODP implementation has good performance compared with other projects. Future research should further develop and confirm these initial findings by using the information to optimize even more the library.

- We tested MACSAD in an X86 environment; future research should evaluate in different platforms as ARMv8, Tofino, this can open a new area with the inclusion of more CPU cores and a dedicate P4 chipset.

- To improve an optimize the LPM implementation, the analysis of the Cache-misses and CPU cycles consumed by the process is an important part of the research. We evaluated our implementation using Perf; future research should explore other tools such as eBPF that can prepare user information in kernel context and transfer only needed information to user space adding more resources and statistics maps.

[2] <https://github.com/cslev/pcap_gen_perftest>

Bibliography

BIONDI, P. *Welcome to Scapy's documentation!* 2008. Available from Internet: <http://scapy.readthedocs.io/en/latest/>. Cited 2 times on pages 18 and 35.

BOSSHART et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, jul. 2014. ISSN 0146-4833. Cited 2 times on pages 17 and 23.

BOTTA, A.; DAINOTTI, A.; PESCAPÉ, A. A tool for the generation of realistic network workload for emerging networking scenarios. *Computer Networks*, Elsevier, v. 56, n. 15, p. 3531–3547, 2012. Cited on page 67.

BRADNER, S.; MCQUAID, J. *Benchmarking Methodology for Network Interconnect Devices*. 1999. RFC 2544. Cited 4 times on pages 28, 68, 71, and 74.

CESEN, F. R.; PATRA, P. G. K.; ROTHENBERG, C. E. Bb-gen: A packet crafter for data plane evaluation. In: SBRC. [S.l.], 2018. Cited on page 67.

CSIKOR, L.; SZALAY, M.; SONKOLY, B.; TOKA, L. *Network Function Performance Analyzer*. 2015. <http://nfpa.tmit.bme.hu>. Cited 3 times on pages 28, 29, and 67.

CSIKOR, L.; SZALAY, M.; SONKOLY, B.; TOKA, L. Nfpa: Network function performance analyzer. *IEEE Conference on Network Function Virtualization and Software Defined Networks Demo Track*, 2015. Cited 4 times on pages 9, 28, 30, and 67.

DANG, H. T.; WANG, H.; JEPSEN, T.; BREBNER, G.; KIM, C.; REXFORD, J.; SOULÉ, R.; WEATHERSPOON, H. Whippersnapper: A p4 language benchmark suite. In: *Proceedings of the Symposium on SDN Research*. New York, NY, USA: ACM, 2017. (SOSR '17), p. 95–101. ISBN 978-1-4503-4947-5. Available from Internet: <http://doi.acm.org/10.1145/3050220.3050231>. Cited on page 36.

DEGERMARK, M.; BRODNIK, A.; CARLSSON, S.; PIN, S. Small forwarding tables for fast routing lookups. *ACM SIGCOMM Computer Communication Review*, 1997. Available from Internet: <http://dl.acm.org/citation.cfm?id=263133>. Cited on page 35.

EATHERTON GEORGE VARGHESE, Z. D. W. Tree bitmap : Hardware/software ip lookups with incremental updates. *ACM SIGCOMM Computer Communications Review*, April 2004. Available from Internet: <http://cseweb.ucsd.edu/~varghese/PAPERS/ccr2004.pdf>. Cited on page 34.

EATHERTONK, W.; VARGHESE, G.; DITTIA, Z. Tree bitmap: hardware/software ip lookups with incremental updates. *ACM SIGCOMM Computer Communications Review*, 2004. Available from Internet: <http://cseweb.ucsd.edu/~varghese/PAPERS/ccr2004.pdf>. Cited on page 35.

FERRAGINA, P.; LUCCIO, F.; MANZINI, G.; MUTHUKRISHNAN, S. Compressing and indexing labeled trees, with applications. *Journal of the ACM (JACM)*, November 2009. Available from Internet: <http://dl.acm.org/citation.cfm?id=1613680>. Cited on page 33.

FIESSLER, A.; HAGER, S.; SCHEUERMANN, B. Flexible line speed network packet classification using hybrid on-chip matching circuits. In: *HPSR '17: Proceedings of 2017 IEEE 18th International Conference on High Performance Switching and Routing*. [S.l.: s.n.], 2017. To appear. Cited on page 35.

GNUPLOT. *Gnuplot*. 1986. <http://www.gnuplot.info/>. Cited on page 29.

GUREVICH, V. *P4 Tutorial*. 2015. <https://p4.org/assets/Nov-2015-P4-Bootcamp-P4-Tutorial.pdf>. Available from Internet: <https://p4.org/assets/Nov-2015-P4-Bootcamp-P4-Tutorial.pdf>. Cited 3 times on pages 9, 22, and 23.

HANNA, M.; CHO, S.; MELHEM, R. A novel scalable ipv6 lookup scheme using compressed pipelined tries. *IFIP 10th international TC 6 conference on Networking - Volume Part I*, May 2011. Available from Internet: <http://dl.acm.org/citation.cfm?id=2008820>. Cited on page 34.

INTEL. *Intel Threading Building Blocks*. 2011. <http://threadingbuildingblocks.org/>. Available from Internet: <http://threadingbuildingblocks.org/>. Cited on page 34.

INTEL. *DPDK: Data Plane Development Kit*. 2014. <http://dpdk.org>. Cited 2 times on pages 28 and 29.

JUNIPER NETWORKS. *How many Packets per Second per port are needed to achieve Wire-Speed?* 2015. Https://kb.juniper.net. Available from Internet: <https://kb.juniper.net/InfoCenter/index?page=content&id=KB14737>. Cited on page 29.

KNUTSSON, K. *RWS Synthetic Pcap Generator*. 2014. Https://github.com/karknu/rws. Available from Internet: <https://github.com/karknu/rws>. Cited on page 35.

KREUTZ, D.; RAMOS, F. M. V.; VERISSIMO, P.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. Software-Defined Networking: A Comprehensive Survey. p. 1–61, 2014. Available from Internet: <http://arxiv.org/abs/1406.0440>. Cited on page 17.

LAKI, S.; HORPáCSI, D.; VöRöS, P.; KITLEI, R.; LESKó, D.; TEJFEL, M. High speed packet forwarding compiled from protocol independent data plane specifications. In: *ACM SIGCOMM'16 Posters and Demos*. [S.l.: s.n.], 2016. Cited on page 35.

LETTIERI, G.; MAFFIONE, V.; RIZZO, L. A survey of fast packet i/o technologies for network function virtualization. In: KUNKEL, J. M.; YOKOTA, R.; TAUFER, M.; SHALF, J. (Ed.). *High Performance Computing*. Cham: Springer International Publishing, 2017. p. 579–590. ISBN 978-3-319-67630-2. Cited on page 76.

MAHALINGAM, M.; DUTT, D.; DUDA, K.; AGARWAL, P.; KREEGER, L.; SRIDHAR, T.; BURSELL, M.; WRIGHT, C. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. [S.l.], 2014. <https://tools.ietf.org/html/rfc7348>. Available from Internet: <https://tools.ietf.org/html/rfc7348>. Cited on page 29.

MCCANNE, S.; JACOBSON, V. The bsd packet filter: A new architecture for user-level packet capture. In: *USENIX winter*. [S.l.: s.n.], 1993. v. 46. Cited on page 80.

MCKEOWN, N. *Programming the Forwarding Plane*. 2016. Https://forum.stanford.edu/events/2016/slides/plenary/Nick.pdf. Available from Internet: <https://forum.stanford.edu/events/2016/slides/plenary/Nick.pdf>. Cited 3 times on pages 9, 22, and 25.

MELO, A. C. D. The new linux'perf'tools. In: *Slides from Linux Kongress*. [S.l.: s.n.], 2010. v. 18. Cited on page 80.

NIKOLENKO, S. I.; KOGANY, K.; RETVARIZ, G.; BERCZI-KOVACS, E. R.; SHALIMOV, A. How to represent ipv6 forwarding tables on ipv4 or mpls dataplanes. *IEEE Computer Communications Workshops (INFOCOM WKSHPS)*, April 2016. Available from Internet: <http://ieeexplore.ieee.org/document/7562132/>. Cited on page 34.

NÖTZLI, A.; KHAN, J.; FINGERHUT, A.; BARRETT, C.; ATHANAS, P. P4pktgen: Automated test case generation for p4 programs. In: ACM. *Proceedings of the Symposium on SDN Research*. [S.l.], 2018. p. 5. Cited on page 36.

OLSSON, R. Pktgen the linux packet generator. In: *Proceedings of the Linux Symposium, Ottawa, Canada*. [S.l.: s.n.], 2005. v. 2, p. 11–24. Cited on page 67.

OPENDATAPLANE. *ODP Helper library*. 2013. <https://github.com/Linaro/odp/tree/master/helper>. Available from Internet: <https://github.com/Linaro/odp/tree/master/helper>. Cited on page 27.

OPENDATAPLANE. *OpenDataPlane DPDK*. 2013. Https://github.com/Linaro/odp-dpdk. Available from Internet: <https://github.com/Linaro/odp-dpdk>. Cited on page 27.

OPENDATAPLANE. *OpenDataPlane.org*. 2013. <https://www.opendataplane.org>. Cited 3 times on pages 9, 17, and 27.

OSTINATO. *Ostinato*. 2010. Https://ostinato.org. Available from Internet: <https://ostinato.org>. Cited on page 35.

OVS. *Open vSwitch*. 2009. <http://openvswitch.org/>. Available from Internet: <http://openvswitch.org/>. Cited on page 33.

P4.ORG. *P4app*. 2013. Https://github.com/p4lang/p4app. Available from Internet: <https://github.com/p4lang/p4app>. Cited on page 36.

PATRA, P. G.; ROTHENBERG, C. E.; PONGRáCZ, G. MACSAD: Multi-Architecture Compiler System for Abstract Dataplanes (Aka Partnering P4 with ODP). In: *ACM SIGCOMM'16 Demo and Poster Session*. [S.l.: s.n.], 2016. ISBN 978-1-4503-4193-6. Cited 4 times on pages 9, 17, 18, and 31.

PATRA, P. G.; ROTHENBERG, C. E.; PONGRACZ, G. Macsad: High performance dataplane applications on the move. In: *IEEE HPSR*. [S.l.: s.n.], 2017. p. 1–6. Cited 2 times on pages 17 and 18.

PATRA, P. G. K.; CESEN, F. E. R.; MEJIA, J. S.; FEFERMAN, D. L.; CSIKOR, L.; ROTHENBERG, C. E.; PONGRACZ, G. Toward a sweet spot of data plane programmability, portability, and performance: On the scalability of multi-architecture p4 pipelines. *IEEE Journal on Selected Areas in Communications*, v. 36, n. 12, p. 2603–2611, Dec 2018. ISSN 0733-8716. Cited on page 18.

PONG, F.; TZENG, N. Concise lookup tables for ipv4 and ipv6 longest prefix matching in scalable routers. *IEEE/ACM Transactions on Networking*, v. 20, n. 3, p. 729–741, June 2012. ISSN 1063-6692. Cited 2 times on pages 18 and 34.

RAVIKUMAR, V.; MAHAPATRA, R. Tcam architecture for ip lookup using prefix properties. *IEEE*, August 2004. Available from Internet: <http://ieeexplore.ieee.org/abstract/document/1289292/>. Cited on page 33.

RIPE NCC. *Visibility of Prefix Lengths in IPv4 and IPv6*. 2010. Https://labs.ripe.net. Available from Internet: <https://labs.ripe.net/Members/dbayer/visibility-of-prefix-lengths>. Cited on page 38.

RIZZO, L. netmap: A novel framework for fast packet i/o. In: *USENIX ATC 12*. [S.l.: s.n.], 2012. ISBN 978-931971-93-5. Cited on page 28.

RODRIGUEZ, F.; PATRA, P. G. K.; CSIKOR, L.; ROTHENBERG, C.; LAKI, P. V. S.; PONGRáCZ, G. Bb-gen: A packet crafter for p4 target evaluation. In: *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 111–113. ISBN 978-1-4503-5915-3. Available from Internet: <http://doi.acm.org/10.1145/3234200.3234229>. Cited on page 67.

ROSEN, E.; VISWANATHAN, A.; CALLON, R. *Multiprotocol Label Switching Architecture*. [S.l.], 2001. <https://tools.ietf.org/html/rfc3031>. Available from Internet: <https://tools.ietf.org/html/rfc3031>. Cited on page 29.

RéTVáRI, G.; TAPOLCAI, J.; KORöSI, A.; MAJDáN, A.; HESZBERGER, Z. Compressing ip forwarding tables: Towards entropy bounds and beyond. *ACM SIGCOMM Computer Communication Review*, August 2013. Available from Internet: <http://conferences.sigcomm.org/sigcomm/2013/papers/sigcomm/p111.pdf>. Cited 3 times on pages 18, 33, and 34.

SHAHHAZ et al. PISCES: A Programmable, Protocol-Independent Software Switch. In: *ACM SIGCOMM*. [S.l.: s.n.], 2016. ISBN 978-1-4503-4193-6. Cited 3 times on pages 18, 25, and 33.

SKLOWER, K. *A tree-based packet routing table for Berkeley UNIX*. [S.l.], 2001. Available from Internet: <https://people.eecs.berkeley.edu/~sklower/routing.pdf>. Cited on page 33.

SRINIVASAN, V.; VARGHESE, G. Faster ip lookups using controlled prefix expansion. *ACM SIGMETRICS Performance Evaluation Review*, 1999. Available from Internet: <http://dl.acm.org/citation.cfm?id=277863>. Cited on page 34.

TCPDUMP. *Tcpdump*. 2010. Https://www.tcpdump.org. Available from Internet: <https://www.tcpdump.org>. Cited on page 67.

TOBOLA, J.; KOřENEK, J. Effective hash-based ipv6 longest prefix match. *IEEE 14th International Symposium Design and Diagnostics of Electronic Circuits and Systems*, May 2011. Available from Internet: <http://ieeexplore.ieee.org/document/5783105/>. Cited on page 34.

TURULL, D.; SJöDIN, P.; OLSSON, R. Pktgen: Measuring performance on high speed networks. *Computer Communications*, 2016. ISSN 0140-3664. Cited on page 29.

WANG, G.; TZENG, N. feng. Tcam-based forwarding engine with minimum independent prefix set (mips) for fast updating. *Communications, 2006. ICC '06. IEEE International Conference on*, December 2006. Available from Internet: <http://ieeexplore.ieee.org/document/4024102/>. Cited on page 35.

WATERLOO, U. of. *Binary trees*. 2018. <https://www.student.cs.uwaterloo.ca/~cs115/coursenotes1/08-bintrees-post.pdf>. Available from Internet: <https://www.student.cs.uwaterloo.ca/~cs115/coursenotes1/08-bintrees-post.pdf>. Cited 2 times on pages 9 and 39.

WILES, K. *pktgen-dpdk*. 2010. <http://dpdk.org/browse/apps/pktgen-dpdk/refs/>. Cited on page 29.

ZHOU, D.; FAN, B.; LIM, H.; KAMINSKY, M.; ANDERSEN, D. G. Scalable, high performance ethernet forwarding with cuckooswitch. *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, December 2013. Available from Internet: <http://dl.acm.org/citation.cfm?id=2535379>. Cited on page 33.

Annex

ANNEX A – Publications

- **F. R. Cesen**, G. P. Patra, and C. E. Rothenberg. BB-Gen: A Packet Crafter for Data Plane Evaluation. In: *XXXVI Simpósio Brasileiros de Redes de Computadores SBRC 2018 Salão de Ferramentas*, Campos do Jordão, SP, Brazil, May 2018.

- **F. R. Cesen**, G. P. Patra, C. E. Rothenberg, and G. Pongrácz. BB-Gen: A Packet Crafter for Performance Evaluation of P4 Data Planes. In: *5th P4 Workshop*, Stanford University, CA, USA, June 2018.

- G. P. patra, **F. R. Cesen**, J. S. Mejia, D. Feferman, C. E. Rothenberg, and G. Pongrácz. MACSAD: An Exemplar Realization of Multi-Architecture P4 Pipelines. In: *5th P4 Workshop*, Stanford University, CA, USA, June 2018.

- **F. R. Cesen**, G. P. Patra, C. E. Rothenberg, and G. Pongrácz. Design, Implementation and Evaluation of IPv4/IPv6 Longest Prefix Match support in P4 Dataplanes. In: *17º Workshop em Desempenho de Sistemas Computacionais e de Comunicação WPerformance 2018*, Natal, RN, Brazil, July 2018.

- **F. R. Cesen**, G. P. Patra, L. Csikor, C. E. Rothenberg, P. Vörös, S. Laki and G. Pongrácz. BB-Gen: A Packet Crafter for P4 Target Evaluation. In: *ACM Special Interest Group on Data Communication SIGCOMM 2018 Posters, Demos, and Student Research Competition*, Budapest, Hungary, August 2018.

- G. P. Patra, **F. R. Cesen**, J. S. Mejia, D. 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. Under Submission in: *IEEE COMSOC JSAC'18 Special Issue on Scalability Issues and Solutions for Software Defined Networks.*

- **F. E. R. Cesen**, and C. R. E. Rothenberg. Design, Implementation and Evaluation of IPv4/IPv6 Longest Prefix Match support in Multi-Architecture Programmable Dataplanes. In: *Décimo Primeiro Encontro dos Alunos e Docentes do Departamento de Engenharia de Computação e Automação Industrial*, Campinas, SP, Brazil, November 2018.

ANNEX B – ODP IPv6 Code

B.1 ODP IPv6 lookup code

```c
#include "config.h"
#include <string.h>
#include <stdint.h>
#include <errno.h>
#include <stdio.h>
#include <odp/helper/odph_iplookuptablev6.h>
#include "odph_list_internal.h"
#include "odph_debug.h"
#include <odp_api.h>
#include <odp/helper/ip.h>

typedef __int128 _uint128_t;
typedef unsigned __int128 uint128_t;

/** @magic word, write to the first byte of the memory block
 *   to indicate this block is used by a ip lookup table
 */
#define ODPH_IP_LOOKUP_TABLE_MAGIC_WORD 0xCFCFFCFC

/* The length(bit) of the IPv6 address */
#define IP_LENGTH 128

/* The number of L1 entries */
#define ENTRY_NUM_L1     (1 << 24)
/* The size of one L2\L3 subtree */
#define ENTRY_NUM_SUBTREE (1 << 12)

#define WHICH_CHILD(ip, cidr) ((ip >> (IP_LENGTH - cidr)) & 0x00000001)

/** @internal entry struct
 *   Structure store an entry of the ip prefix table.
 *   Because of the leaf pushing, each entry of the table must have
 *   either a child entry, or a nexthop info.
 *   If child == 0 and index != ODP_BUFFER_INVALID, this entry has
 *    a nexthop info, index indicates the buffer that stores the
 *    nexthop value, and ptr points to the address of the buffer.
 *   If child == 1, this entry has a subtree, index indicates
 *    the buffer that stores the subtree, and ptr points to the
 *    address of the buffer.
 */
```

```c
typedef struct {
  union {
    odp_buffer_t nexthop;
    void *ptr;
  };
  union {
    _uint128_t u8;
    struct {
#if ODP_BYTE_ORDER == ODP_BIG_ENDIAN
      uint8_t child : 1;
      uint8_t cidr  : 7;
#else
      uint8_t cidr  : 7;
      uint8_t child : 1;
#endif
    };
  };
} prefix_entry_t;

#define ENTRY_SIZE (sizeof(prefix_entry_t) + sizeof(odp_buffer_t))
#define ENTRY_BUFF_ARR(x) ((odp_buffer_t *)(void *)((char *)x \
      + sizeof(prefix_entry_t) * ENTRY_NUM_SUBTREE))

/** @internal trie node struct
 *  In this IP lookup algorithm, we use a
 *  binary tire to detect the overlap prefix.
 */
typedef struct trie_node {
  /* tree structure */
  struct trie_node *parent;
  struct trie_node *left;
  struct trie_node *right;
  /* IP prefix length */
  uint8_t cidr;
  /* Nexthop buffer index */
  odp_buffer_t nexthop;
  /* Buffer that stores this node */
  odp_buffer_t buffer;
} trie_node_t;

/** Number of L2\L3 entries(subtrees) per cache cube. */
#define CACHE_NUM_SUBTREE (1 << 13)
/** Number of trie nodes per cache cube. */
#define CACHE_NUM_TRIE    (1 << 20)

/** @typedef cache_type_t
 *  Cache node type
 */
typedef enum {
  CACHE_TYPE_SUBTREE = 0,
  CACHE_TYPE_TRIE
} cache_type_t;

/** A IP lookup table structure. */
typedef struct {
  /**< for check */
  uint32_t magicword;
  /** Name of the hash. */
  char name[ODPH_TABLE_NAME_LEN];
  /** Total L1 entries. */
  prefix_entry_t *l1e;
  /** Root node of the binary trie */
  trie_node_t *trie;
  /** Length of value. */
  uint32_t nexthop_len;
  /** Queues of free slots (caches)
   *  There are two queues:
   *  - free_slots[CACHE_TYPE_SUBTREE] is used for L2 and
   *    L3 entries (subtrees). Each entry stores an 8-bit
   *    subtree.
   *  - free_slots[CACHE_TYPE_TRIE] is used for the binary
   *    trie. Each entry contains a trie node.
   */
  odp_queue_t free_slots[2];
  /** The number of pool used by each queue. */
  uint32_t cache_count[2];
} odph_iplookup_table_impl ODP_ALIGNED_CACHE;

/**********************************************************
 ****************   Cache management   *******************
 **********************************************************/

/** Destroy all caches */
static void cache_destroy(odph_iplookup_table_impl *impl){
  odp_queue_t queue;
  odp_event_t ev;
  uint32_t i = 0, count = 0;
  char pool_name[ODPH_TABLE_NAME_LEN + 8];

  /* free all buffers in the queue */
  for (; i < 2; i++) {
    queue = impl->free_slots[i];
    if (queue == ODP_QUEUE_INVALID)
      continue;

    while ((ev = odp_queue_deq(queue))
        != ODP_EVENT_INVALID) {
      odp_buffer_free(odp_buffer_from_event(ev));
    }
    odp_queue_destroy(queue);
  }

  /* destroy all cache pools */
  for (i = 0; i < 2; i++) {
    for (count = 0; count < impl->cache_count[i]; count++) {
      sprintf(
          pool_name, "%s_%d_%d",
          impl->name, i, count);
      odp_pool_destroy(odp_pool_lookup(pool_name));
    }
  }
}

/** According to the type of cahce, set the value of
 *  a buffer to the initial value.
 */
static void cache_init_buffer(odp_buffer_t buffer, cache_type_t type,
    uint32_t size){
  int i = 0;
  void *addr = odp_buffer_addr(buffer);

  memset(addr, 0, size);
  if (type == CACHE_TYPE_SUBTREE) {
    prefix_entry_t *entry = (prefix_entry_t *)addr;

    for (i = 0; i < ENTRY_NUM_SUBTREE; i++, entry++)
      entry->nexthop = ODP_BUFFER_INVALID;
  } else if (type == CACHE_TYPE_TRIE) {
    trie_node_t *node = (trie_node_t *)addr;

    node->buffer = buffer;
    node->nexthop = ODP_BUFFER_INVALID;
  }
}

/** Create a new buffer pool, and insert its buffer into the queue. */
static int cache_alloc_new_pool(odph_iplookup_table_impl *tbl, cache_type_t
     type){
  odp_pool_t pool;
  odp_pool_param_t param;
  odp_queue_t queue = tbl->free_slots[type];

  odp_buffer_t buffer;
  char pool_name[ODPH_TABLE_NAME_LEN + 8];
  uint32_t size = 0, num = 0;

  /* Create new pool (new free buffers). */
  odp_pool_param_init(&param);
  param.type = ODP_POOL_BUFFER;
  param.buf.align = ODP_CACHE_LINE_SIZE;
  if (type == CACHE_TYPE_SUBTREE) {
    num = CACHE_NUM_SUBTREE;
    size = ENTRY_SIZE * ENTRY_NUM_SUBTREE;
  } else if (type == CACHE_TYPE_TRIE) {
    num = CACHE_NUM_TRIE;
    size = sizeof(trie_node_t);
  } else {
    ODPH_DBG("wrong cache_type_t.\n");
    return -1;
  }
  param.buf.size = size;
  param.buf.num = num;

  sprintf(
      pool_name, "%s_%d_%d",
      tbl->name, type, tbl->cache_count[type]);
  pool = odp_pool_create(pool_name, &param);
  if (pool == ODP_POOL_INVALID) {
    ODPH_DBG("failed to create a new pool.\n");
    return -1;
  }

  /* insert new free buffers into queue */
  while ((buffer = odp_buffer_alloc(pool))
      != ODP_BUFFER_INVALID) {
    cache_init_buffer(buffer, type, size);
    odp_queue_enq(queue, odp_buffer_to_event(buffer));
  }

  tbl->cache_count[type]++;
  return 0;
}

/** Get a new buffer from a cache list. If there is no
 *  available buffer, allocate a new pool.
 */
static odp_buffer_t cache_get_buffer(odph_iplookup_table_impl *tbl,
    cache_type_t type){
```

```
  odp_buffer_t buffer = ODP_BUFFER_INVALID;
  odp_queue_t queue = tbl->free_slots[type];

  /* get free buffer from queue */
  buffer = odp_buffer_from_event(
        odp_queue_deq(queue));

  /* If there is no free buffer available, allocate new pool */
  if (buffer == ODP_BUFFER_INVALID) {
    cache_alloc_new_pool(tbl, type);
    buffer = odp_buffer_from_event(odp_queue_deq(queue));
  }

  return buffer;
}

/*********************************************************
 *****************      Binary trie      *******************
 *********************************************************/

/* Initialize the root node of the trie */
static int trie_init(odph_iplookup_table_impl *tbl){
  trie_node_t *root = NULL;
  odp_buffer_t buffer = cache_get_buffer(tbl, CACHE_TYPE_TRIE);

  if (buffer != ODP_BUFFER_INVALID) {
    root = (trie_node_t *)odp_buffer_addr(buffer);
    root->cidr = 0;
    tbl->trie = root;
    return 0;
  }

  return -1;
}

/* Destroy the whole trie (recursively) */
static void trie_destroy(odph_iplookup_table_impl *tbl, trie_node_t *trie){
  if (trie->left != NULL)
    trie_destroy(tbl, trie->left);
  if (trie->right != NULL)
    trie_destroy(tbl, trie->right);

  /* destroy this node */
  odp_queue_enq(
      tbl->free_slots[CACHE_TYPE_TRIE],
      odp_buffer_to_event(trie->buffer));
}
```

```c

/* Insert a new prefix node into the trie
 * If the node is already existed, update its nexthop info,
 *   Return 0 and set nexthop pointer to INVALID.
 * If the node is not exitsed, create this target node and
 *   all nodes along the path from root to the target node.
 *   Then return 0 and set nexthop pointer points to the
 *   new buffer.
 * Return -1 for error.
 */
static int trie_insert_node(
    odph_iplookup_table_impl *tbl, trie_node_t *root,
    _uint128_t ip, uint8_t cidr, odp_buffer_t nexthop)
{
  uint8_t level = 0, child;
  odp_buffer_t buf;
  trie_node_t *node = root, *prev = root;

  /* create/update all nodes along the path
   * from root to the new node. */
  for (level = 1; level <= cidr; level++) {
    child = WHICH_CHILD(ip, level);

    node = child == 0 ? prev->left : prev->right;
    /* If the child node doesn't exit, create it. */
    if (node == NULL) {
      buf = cache_get_buffer(tbl, CACHE_TYPE_TRIE);
      if (buf == ODP_BUFFER_INVALID)
        return -1;

      node = (trie_node_t *)odp_buffer_addr(buf);
      node->cidr = level;
      node->parent = prev;

      if (child == 0)
        prev->left = node;
      else
        prev->right = node;
    }
    prev = node;
  }

  /* The final one is the target. */
  node->nexthop = nexthop;
  return 0;
}

```

```
/* Delete a node */
static int trie_delete_node(
    odph_iplookup_table_impl *tbl,
    trie_node_t *root, _uint128_t ip, uint8_t cidr)
{
  if (root == NULL)
    return -1;

  /* The default prefix (root node) cannot be deleted. */
  if (cidr == 0)
    return -1;

  trie_node_t *node = root, *prev = NULL;
  uint8_t level = 1, child = 0;
  odp_buffer_t tmp;

  /* Find the target node. */
  for (level = 1; level <= cidr; level++) {
    child = WHICH_CHILD(ip, level);
    node = (child == 0) ? node->left : node->right;
    if (node == NULL) {
      ODPH_DBG("Trie node is not existed\n");
      return -1;
    }
  }

  node->nexthop = ODP_BUFFER_INVALID;

  /* Delete all redundant nodes along the path. */
  for (level = cidr; level > 0; level--) {
    if (
      node->left != NULL || node->right != NULL ||
      node->nexthop != ODP_BUFFER_INVALID)
      break;

    child = WHICH_CHILD(ip, level);
    prev = node->parent;

    /* free trie node */
    tmp = node->buffer;
    cache_init_buffer(
        tmp, CACHE_TYPE_TRIE, sizeof(trie_node_t));
    odp_queue_enq(
        tbl->free_slots[CACHE_TYPE_TRIE],
        odp_buffer_to_event(tmp));

    if (child == 0)
```

```
      prev->left = NULL;
    else
      prev->right = NULL;
    node = prev;
  }
  return 0;
}

/* Detect the longest overlapping prefix. */
static int trie_detect_overlap(
    trie_node_t *trie, _uint128_t ip, uint8_t cidr,
    uint8_t leaf_push, uint8_t *over_cidr,
    odp_buffer_t *over_nexthop)
{
  uint8_t child = 0;
  uint32_t level, limit = cidr > leaf_push ? leaf_push + 1 : cidr;
  trie_node_t *node = trie, *longest = trie;

  for (level = 1; level < limit; level++) {
    child = WHICH_CHILD(ip, level);
    node = (child == 0) ? node->left : node->right;
    if (node->nexthop != ODP_BUFFER_INVALID)
      longest = node;
  }

  *over_cidr = longest->cidr;
  *over_nexthop = longest->nexthop;
  return 0;
}

/*****************************************************
 **************   IP prefix lookup table   ***************
 ******************************************************/

odph_table_t odph_iplookupv6_table_lookup(const char *name) {
  odph_iplookup_table_impl *tbl = NULL;
  odp_shm_t shm;

  if (name == NULL || strlen(name) >= ODPH_TABLE_NAME_LEN)
    return NULL;

  shm = odp_shm_lookup(name);
  if (shm != ODP_SHM_INVALID)
    tbl = (odph_iplookup_table_impl *)odp_shm_addr(shm);

  if (
    tbl != NULL &&
    tbl->magicword == ODPH_IP_LOOKUP_TABLE_MAGIC_WORD &&
    strcmp(tbl->name, name) == 0)
    return (odph_table_t)tbl;

  return NULL;
}

odph_table_t odph_iplookupv6_table_create(const char *name,
          uint32_t p1 ODP_UNUSED,
          uint32_t p2 ODP_UNUSED,
          uint32_t value_size)
{
  odph_iplookup_table_impl *tbl;
  odp_shm_t shm_tbl;
  odp_queue_t queue;
  odp_queue_param_t qparam;
  unsigned i;
  uint32_t impl_size, l1_size;
  char queue_name[ODPH_TABLE_NAME_LEN + 2];

  /* Check for valid parameters */
  if (strlen(name) == 0) {
    ODPH_DBG("invalid parameters\n");
    return NULL;
  }

  /* Guarantee there's no existing */
  tbl = (void *)odph_iplookupv6_table_lookup(name);
  if (tbl != NULL) {
    ODPH_DBG("IP prefix table %s already exists\n", name);
    return NULL;
  }

  /* Calculate the sizes of different parts of IP prefix table */
  impl_size = sizeof(odph_iplookup_table_impl);
  l1_size = ENTRY_SIZE * ENTRY_NUM_L1;

  shm_tbl = odp_shm_reserve(
        name, impl_size + l1_size,
        ODP_CACHE_LINE_SIZE, ODP_SHM_SW_ONLY);

  if (shm_tbl == ODP_SHM_INVALID) {
    ODPH_DBG(
      "shm allocation failed for odph_iplookup_table_impl %s\n",
      name);
    return NULL;
  }

  tbl = (odph_iplookup_table_impl *)odp_shm_addr(shm_tbl);
  memset(tbl, 0, impl_size + l1_size);

  /* header of this mem block is the table impl struct,
   * then the l1 entries array.
   */
  tbl->l1e = (prefix_entry_t *)(void *)((char *)tbl + impl_size);
  for (i = 0; i < ENTRY_NUM_L1; i++)
    tbl->l1e[i].nexthop = ODP_BUFFER_INVALID;

  /* Setup table context. */
  snprintf(tbl->name, sizeof(tbl->name), "%s", name);
  tbl->magicword = ODPH_IP_LOOKUP_TABLE_MAGIC_WORD;
  tbl->nexthop_len = value_size;

  /* Initialize cache */
  for (i = 0; i < 2; i++) {
    tbl->cache_count[i] = 0;

    odp_queue_param_init(&qparam);
    qparam.type = ODP_QUEUE_TYPE_PLAIN;
    sprintf(queue_name, "%s_%d", name, i);
    queue = odp_queue_create(queue_name, &qparam);
    if (queue == ODP_QUEUE_INVALID) {
      ODPH_DBG("failed to create queue");
      cache_destroy(tbl);
      return NULL;
    }
    tbl->free_slots[i] = queue;
    cache_alloc_new_pool(tbl, i);
  }

  /* Initialize tire */
  if (trie_init(tbl) < 0) {
    odp_shm_free(shm_tbl);
    return NULL;
  }

  return (odph_table_t)tbl;
}

int odph_iplookupv6_table_destroy(odph_table_t tbl) {
  int i, j;
  odph_iplookup_table_impl *impl = NULL;
  prefix_entry_t *subtree = NULL;
  odp_buffer_t *buff1 = NULL, *buff2 = NULL;

  if (tbl == NULL)
    return -1;

  impl = (odph_iplookup_table_impl *)(void *)tbl;

  /* check magic word */
  if (impl->magicword != ODPH_IP_LOOKUP_TABLE_MAGIC_WORD) {
    ODPH_DBG("wrong magicword for IP prefix table\n");
    return -1;
  }

  /* destroy trie */
  trie_destroy(impl, impl->trie);

  /* free all L2 and L3 entries */
  buff1 = ENTRY_BUFF_ARR(impl->l1e);
  for (i = 0; i < ENTRY_NUM_L1; i++) {
    if ((impl->l1e[i]).child == 0)
      continue;

    subtree = (prefix_entry_t *)impl->l1e[i].ptr;
    buff2 = ENTRY_BUFF_ARR(subtree);
    /* destroy all l3 subtrees of this l2 subtree */
    for (j = 0; j < ENTRY_NUM_SUBTREE; j++) {
      if (subtree[j].child == 0)
        continue;
      odp_queue_enq(
          impl->free_slots[CACHE_TYPE_TRIE],
          odp_buffer_to_event(buff2[j]));
    }
    /* destroy this l2 subtree */
    odp_queue_enq(
        impl->free_slots[CACHE_TYPE_TRIE],
        odp_buffer_to_event(buff1[i]));
  }

  /* destroy all cache */
  cache_destroy(impl);

  /* free impl */
  odp_shm_free(odp_shm_lookup(impl->name));
  return 0;
}

/* Insert the prefix into level x
 * Return:
```

```
 *   -1 error
 *    0  the table is unmodified
 *    1  the table is modified
 */
static int prefix_insert_into_lx(
    odph_iplookup_table_impl *tbl, prefix_entry_t *entry,
    uint8_t cidr, odp_buffer_t nexthop, uint8_t level)
{
  uint8_t ret = 0;
  uint32_t i = 0, limit = (1 << (level - cidr));
  prefix_entry_t *e = entry, *ne = NULL;

  for (i = 0; i < limit; i++, e++) {
    if (e->child == 1) {
      if (e->cidr > cidr)
        continue;

      e->cidr = cidr;
      /* push to next level */
      ne = (prefix_entry_t *)e->ptr;
      ret = prefix_insert_into_lx(
          tbl, ne, cidr, nexthop, cidr + 8);
    } else {
      if (e->cidr > cidr)
        continue;

      e->child = 0;
      e->cidr = cidr;
      e->nexthop = nexthop;
      ret = 1;
    }
  }
  return ret;
}

static int prefix_insert_iter(
    odph_iplookup_table_impl *tbl, prefix_entry_t *entry,
    odp_buffer_t *buff, __uint128_t ip, uint8_t cidr,
    odp_buffer_t nexthop, uint8_t level, uint8_t depth)
{
  uint8_t state = 0;
  prefix_entry_t *ne = NULL;
  odp_buffer_t *nbuff = NULL;

  /* If child subtree is existed, get it. */
  if (entry->child) {
    //printf("child existed, get it. \n");
    ne = (prefix_entry_t *)entry->ptr;
    nbuff = ENTRY_BUFF_ARR(ne);
  } else {
    /* If the child is not existed, create a new subtree. */
    odp_buffer_t buf, push = entry->nexthop;

    buf = cache_get_buffer(tbl, CACHE_TYPE_SUBTREE);
    if (buf == ODP_BUFFER_INVALID) {
      ODPH_DBG("failed to get subtree buffer from cache.\n");
      return -1;
    }
    ne = (prefix_entry_t *)odp_buffer_addr(buf);
    nbuff = ENTRY_BUFF_ARR(ne);

    entry->child = 1;
    entry->ptr = ne;
    *buff = buf;
    /* If this entry contains a nexthop and a small cidr,
     * push it to the next level.
     */
    if (entry->cidr > 0)
      (void)prefix_insert_into_lx(tbl, ne, entry->cidr,
                   push, entry->cidr + 8);
  }
  ne += (ip >> 120);
  nbuff += (ip >> 120);
  if (cidr <= 8) {
    state = prefix_insert_into_lx(
        tbl, ne, cidr + depth * 8, nexthop, level);
  } else {
    state = prefix_insert_iter(
        tbl, ne, nbuff, ip << 8, cidr - 8,
        nexthop, level + 8, depth + 1);
  }

  return state;
}

int odph_iplookupv6_table_put_value(odph_table_t tbl, void *key, void *
    value) {
  odph_iplookup_table_impl *impl = (void *)tbl;
  odph_iplookupv6_prefix_t *prefix = (odph_iplookupv6_prefix_t *)key;
  prefix_entry_t *l1e = NULL;
  odp_buffer_t nexthop;
  int ret = 0;
  _uint128_t lkp_ip = 0;

  if ((tbl == NULL) || (key == NULL) || (value == NULL))
    return -1;
  nexthop = *((odp_buffer_t *)value);

  if (prefix->cidr == 0)
    return -1;

  ret = odph_ipv6_addr_parse(&lkp_ip, "ffff:ffff:ffff:ffff:ffff:ffff:ffff:
   ffff");
  if (ret < 0) {
    printf("Failed to get IPv6 addr from str\n");
    return -1;
  }

  prefix->ip = prefix->ip & (lkp_ip << (IP_LENGTH - prefix->cidr));
  ret = trie_insert_node(
        impl, impl->trie,
        prefix->ip, prefix->cidr, nexthop);

  if (ret < 0) {
    ODPH_DBG("failed to insert into trie\n");
    return -1;
  }

  l1e = &impl->l1e[prefix->ip >> 112];
  odp_buffer_t *buff = ENTRY_BUFF_ARR(impl->l1e) + (prefix->ip >> 112);

  if (prefix->cidr <= 16) {
    ret = prefix_insert_into_lx(
        impl, l1e, prefix->cidr, nexthop, 16);
  } else {
    ret = prefix_insert_iter(
        impl, l1e, buff,
        ((prefix->ip) << 112), prefix->cidr - 16,
        nexthop, 24, 2);
  }
  return ret;
}

int odph_iplookupv6_table_get_value(odph_table_t tbl, void *key,
          void *buffer ODP_UNUSED,
          uint32_t buffer_size ODP_UNUSED)
{
  odph_iplookup_table_impl *impl = (void *)tbl;
  _uint128_t ip;
  prefix_entry_t *entry;
  odp_buffer_t *buff = (odp_buffer_t *)buffer;

  if ((tbl == NULL) || (key == NULL) || (buffer == NULL))
    return -EINVAL;
  ip = *((_uint128_t *)key);
  entry = &impl->l1e[ip >> 112];
  if (entry == NULL) {
    ODPH_DBG("failed to get L1 entry.\n");
    return -1;
  }
  ip <<= 112;

  while (entry->child) {
    entry = (prefix_entry_t *)entry->ptr;
    entry += ip >> 24;
    ip <<= 8;
  }
  if (entry->nexthop == ODP_BUFFER_INVALID) {
    /* ONLY match the default prefix */
    printf("only match the default prefix\n");
    *buff = ODP_BUFFER_INVALID;
  } else {
    *buff = entry->nexthop;
  }

  return 0;
}

static int prefix_delete_lx(
    odph_iplookup_table_impl *tbl, prefix_entry_t *l1e,
    odp_buffer_t *buff, uint8_t cidr, uint8_t over_cidr,
    odp_buffer_t over_nexthop, uint8_t level)
{
  uint8_t ret, flag = 1;
  prefix_entry_t *e = l1e;
  odp_buffer_t *b = buff;
  uint32_t i = 0, limit = 1 << (level - cidr);

  for (i = 0; i < limit; i++, e++, b++) {
    if (e->child == 1) {
      if (e->cidr > cidr) {
        flag = 0;
        continue;
      }

      prefix_entry_t *ne = (prefix_entry_t *)e->ptr;
      odp_buffer_t *nbuff = ENTRY_BUFF_ARR(ne);

      e->cidr = over_cidr;
      ret = prefix_delete_lx(
          tbl, ne, nbuff, cidr, over_cidr,
          over_nexthop, cidr + 8);

      /* If ret == 1, the next 2^8 entries equal to
       * (over_cidr, over_nexthop). In this case, we
       * should not push the (over_cidr, over_nexthop)
       * to the next level. In fact, we should recycle
       * the next 2^8 entries.
       */
      if (ret) {
        /* destroy subtree */
        cache_init_buffer(
          *b, CACHE_TYPE_SUBTREE,
          ENTRY_SIZE * ENTRY_NUM_SUBTREE);
        odp_queue_enq(
          tbl->free_slots[CACHE_TYPE_SUBTREE],
          odp_buffer_to_event(*b));
        e->child = 0;
        e->nexthop = over_nexthop;
      } else {
        flag = 0;
      }
    } else {
      if (e->cidr > cidr) {
        flag = 0;
        continue;
      } else {
        e->cidr = over_cidr;
        e->nexthop = over_nexthop;
      }
    }
  }
  return flag;
}

/* Check if the entry can be recycled.
 * An entry can be recycled duo to two reasons:
 * - all children of the entry are the same,
 * - all children of the entry have a cidr smaller than the level
 *   bottom bound.
 */
static uint8_t can_recycle(prefix_entry_t *e, uint32_t level) {
  uint8_t recycle = 1;
  int i = 1;
  prefix_entry_t *ne = (prefix_entry_t *)e->ptr;

  if (ne->child)
    return 0;

  uint8_t cidr = ne->cidr;
  odp_buffer_t index = ne->nexthop;

  if (cidr > level)
    return 0;

  ne++;
  for (; i < 256; i++, ne++) {
    if (
        ne->child != 0 || ne->cidr != cidr ||
        ne->nexthop != index) {
      recycle = 0;
      break;
    }
  }
  return recycle;
}

static uint8_t prefix_delete_iter(
    odph_iplookup_table_impl *tbl, prefix_entry_t *e,
    odp_buffer_t *buff, _uint128_t ip, uint8_t cidr,
    uint8_t level, uint8_t depth)
{
  uint8_t ret = 0, over_cidr;
  odp_buffer_t over_nexthop;

  trie_detect_overlap(
      tbl->trie, ip, cidr + 8 * depth, level,
      &over_cidr, &over_nexthop);
  if (cidr > 8) {
    prefix_entry_t *ne =
      (prefix_entry_t *)e->ptr;
    odp_buffer_t *nbuff = ENTRY_BUFF_ARR(ne);

    ne += ((_uint128_t)(ip << level) >> 24);
    nbuff += ((_uint128_t)(ip << level) >> 24);
    ret = prefix_delete_iter(
        tbl, ne, nbuff, ip, cidr - 8,
        level + 8, depth + 1);

    if (ret && can_recycle(e, level)) {
      cache_init_buffer(
        *buff, CACHE_TYPE_SUBTREE,
        ENTRY_SIZE * ENTRY_NUM_SUBTREE);
      odp_queue_enq(
        tbl->free_slots[CACHE_TYPE_SUBTREE],
        odp_buffer_to_event(*buff));
      e->child = 0;
      e->nexthop = over_nexthop;
      e->cidr = over_cidr;
      return 1;
    }
    return 0;
  }

  ret = prefix_delete_lx(
      tbl, e, buff, cidr + 8 * depth,
      over_cidr, over_nexthop, level);
  return ret;
}

int odph_iplookupv6_table_remove_value(odph_table_t tbl, void *key) {
  odph_iplookup_table_impl *impl = (void *)tbl;
  odph_iplookupv6_prefix_t *prefix = (odph_iplookupv6_prefix_t *)key;
  _uint128_t ip;
  uint8_t cidr;

  if ((tbl == NULL) || (key == NULL))
    return -EINVAL;

  ip   = prefix->ip;
  cidr = prefix->cidr;

  if (cidr == 0)
    return -EINVAL;

  prefix_entry_t *entry = &impl->l1e[ip >> 112];
  odp_buffer_t *buff = ENTRY_BUFF_ARR(impl->l1e) + (ip >> 112);
  uint8_t over_cidr, ret;
  odp_buffer_t over_nexthop;

  trie_detect_overlap(
      impl->trie, ip, cidr, 16, &over_cidr, &over_nexthop);

  if (cidr <= 16) {
    prefix_delete_lx(
      impl, entry, buff, cidr, over_cidr, over_nexthop, 16);
  } else {
    prefix_entry_t *ne = (prefix_entry_t *)entry->ptr;
    odp_buffer_t *nbuff = ENTRY_BUFF_ARR(ne);

    ne += ((_uint128_t)(ip << 112) >> 24);
    nbuff += ((_uint128_t)(ip << 112) >> 24);
    ret = prefix_delete_iter(impl, ne, nbuff, ip, cidr - 16, 24, 2);

    if (ret && can_recycle(entry, 16)) {
      cache_init_buffer(
        *buff, CACHE_TYPE_SUBTREE,
        sizeof(prefix_entry_t) * ENTRY_NUM_SUBTREE);
      odp_queue_enq(
        impl->free_slots[CACHE_TYPE_SUBTREE],
        odp_buffer_to_event(*buff));
      entry->child = 0;
      entry->cidr = over_cidr;
      entry->nexthop = over_nexthop;
    }
  }

  return trie_delete_node(impl, impl->trie, ip, cidr);
}

odph_table_ops_t odph_iplookupv6_table_ops = {
  odph_iplookupv6_table_create,
  odph_iplookupv6_table_lookup,
  odph_iplookupv6_table_destroy,
  odph_iplookupv6_table_put_value,
  odph_iplookupv6_table_get_value,
  odph_iplookupv6_table_remove_value
};
```

Listing B.1 – ODP IPv6 lookup code

B.2 ODP IPv6 Parse code

```
#include "config.h"
#include <odp/helper/ip.h>
#include <stdio.h>
#include <string.h>

typedef __int128 _uint128_t;
typedef unsigned __int128 uint128_t;

int odph_ipv4_addr_parse(uint32_t *ip_addr, const char *str) {
  unsigned byte[ODPH_IPV4ADDR_LEN];
  int i;

  memset(byte, 0, sizeof(byte));

```

```c
  if (sscanf(str, "%u.%u.%u.%u",
      &byte[0], &byte[1], &byte[2], &byte[3]) != ODPH_IPV4ADDR_LEN)
    return -1;

  for (i = 0; i < ODPH_IPV4ADDR_LEN; i++)
    if (byte[i] > 255)
      return -1;

  *ip_addr = byte[0] << 24 | byte[1] << 16 | byte[2] << 8 | byte[3];

  return 0;
}

int odph_ipv6_addr_parse(_uint128_t *ip_addr, const char *str){
  unsigned byte[ODPH_IPV6ADDR_LEN];
  int i;
  _uint128_t p_ip1, p_ip2, p_ip3, p_ip4;

  memset(byte, 0, sizeof(byte));

  if (sscanf(str, "%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x",
      &byte[0], &byte[1], &byte[2], &byte[3], &byte[4], &byte[5], &byte[6], &byte[7], &byte[8], &byte[9],
      &byte[10], &byte[11], &byte[12], &byte[13], &byte[14], &byte[15]) != ODPH_IPV6ADDR_LEN)
    return -1;

  for (i = 0; i < ODPH_IPV6ADDR_LEN; i++)
    if (byte[i] > 255)
      return -1;

  p_ip1 = byte[0] << 24 | byte[1] << 16 | byte[2] << 8 | byte[3];
  p_ip2 = byte[4] << 24 | byte[5] << 16 | byte[6] << 8 | byte[7];
  p_ip3 = byte[8] << 24 | byte[9] << 16 | byte[10] << 8 | byte[11];
  p_ip4 = byte[12] << 24 | byte[13] << 16 | byte[14] << 8 | byte[15];
  *ip_addr = p_ip1 << 96 | p_ip2 << 64 |  p_ip3 << 32 | p_ip4;

  return 0;
}
```

Listing B.2 – ODP IPv6 parse code

ANNEX C – Controller Code

C.1 MACSAD IPv4 Controller code

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

#define MAX_MACS 2000000

controller c;

void fill_ipv4_fib_lpm_table(uint8_t ip[4], uint8_t port, uint8_t mac[6])
{
  char buffer[2048];
  struct p4_header* h;
  struct p4_add_table_entry* te;
  struct p4_action* a;
  struct p4_action_parameter* ap,* ap2;
  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, "ipv4_fib_lpm");

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

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

  ap = add_p4_action_parameter(h, a, 2048);
  strcpy(ap->name, "dmac");
  memcpy(ap->bitmap, mac, 6);
  ap->length = 6*8+0;

  ap2 = add_p4_action_parameter(h, a, 2048);
  strcpy(ap2->name, "port");
  ap2->bitmap[0] = port;
  ap2->bitmap[1] = 0;
  ap2->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);
  netconv_p4_action_parameter(ap2);

  send_p4_msg(c, buffer, 2048);
}

void fill_sendout_table(uint8_t port, uint8_t smac[6])
{
    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, "sendout");

    exact = add_p4_field_match_exact(te, 2048);
    strcpy(exact->header.name, "standard_metadata.egress_port");
    exact->bitmap[0] = port;
  exact->bitmap[1] = 0;
    exact->length = 2*8+0;

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

  ap = add_p4_action_parameter(h, a, 2048);
  strcpy(ap->name, "smac");
  memcpy(ap->bitmap, smac, 6);
  ap->length = 6*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);
}
```

```c

uint8_t macs[MAX_MACS][6];
uint8_t portmap[MAX_MACS];
uint8_t ips[MAX_MACS][4];
int mac_count = -1;

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

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

  while (fgets(line, sizeof(line), f)) {
    line[strlen(line)-1] = '\0';
    //TODO why %c?
    if (11 == sscanf(line, "%d.%d.%d.%d %x:%x:%x:%x:%x:%x %d",
          &values_ip[0], &values_ip[1], &values_ip[2], &values_ip[3],
          &values[0], &values[1], &values[2],
          &values[3], &values[4], &values[5], &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];
      portmap[mac_count] = (uint8_t) port;

    } else {
      printf("Wrong format error in line %d : %s\n", mac_count+2, line);
      fclose(f);
      return -1;
    }
  }
  fclose(f);
  return 0;
}
```

```

void dhf(void* b) {
  printf("Unknown digest received\n");
}

void init() {
  int i;
  uint8_t smac[6] = {0xd0, 0x69, 0x0f, 0xa8, 0x39, 0x90};
  printf("INIT");
  clock_t begin = clock();
  for (i=0;i<=mac_count;++i)
  {
    fill_ipv4_fib_lpm_table(ips[i], portmap[i], macs[i]);

        if (0 == (i%1000)){
      printf("%d inside sleep \n",i);
      sleep(1);
    }
    fill_sendout_table(portmap[i], smac);
        usleep(1000);
  }

  clock_t end = clock();
  double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
  printf ("ctrl Total entries sent %d time %f\n",i,time_spent);

}

int main(int argc, char* argv[])
{
  uint8_t ip[4] = {192,168,1,1};
    uint8_t mac[6] = {0xa0, 0x36, 0x9f, 0x3e, 0x94, 0xea};
  uint8_t port = 1;

  uint8_t ip2[4] = {192,168,0,1};
    uint8_t mac2[6] = {0xa0, 0x36, 0x9f, 0x3e, 0x94, 0xe8};
  uint8_t port2 = 0;

    uint8_t smac[6] = {0xd0, 0x69, 0x0f, 0xa8, 0x39, 0x90};


  if (argc>1) {
    if (argc!=2) {
      printf("Too many arguments...\nUsage: %s <filename(optional)>\n",
   argv[0]);
      return -1;
    }
    printf("Command line argument is present...\nLoading configuration data
   ...\n");
    if (read_macs_and_ports_from_file(argv[1])<0) {
      printf("File cannot be opened...\n");
      return -1;
    }
  }

  printf("Create and configure l3 test controller...\n");
  c = create_controller_with_init(11111, 3, dhf, init);
  fill_ipv4_fib_lpm_table(ip, port, mac);
  fill_ipv4_fib_lpm_table(ip2, port2, mac2);

  fill_sendout_table(port, smac);
  fill_sendout_table(port2, smac);

  printf("Launching controller's main loop...\n");
  execute_controller(c);

  printf("Destroy controller\n");
  destroy_controller(c);

  return 0;
}
```

Listing C.1 – MACSAD IPv4 Controller code

C.2 MACSAD IPv6 Controller code

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

#define MAX_MACS 2000000

controller c;

void fill_ipv6_fib_lpm_table(uint8_t ip[16], uint8_t port, uint8_t mac[6])
{
  char buffer[2048];
  struct p4_header* h;
  struct p4_add_table_entry* te;
  struct p4_action* a;
  struct p4_action_parameter* ap,* ap2;
  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, "ipv6_fib_lpm");

  exact = add_p4_field_match_exact(te, 2048);
  strcpy(exact->header.name, "ipv6.dstAddr");
  memcpy(exact->bitmap, ip, 16);
  exact->length = 16*8+0;

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

  ap = add_p4_action_parameter(h, a, 2048);
  strcpy(ap->name, "dmac");
  memcpy(ap->bitmap, mac, 6);
  ap->length = 6*8+0;

  ap2 = add_p4_action_parameter(h, a, 2048);
  strcpy(ap2->name, "port");
  ap2->bitmap[0] = port;
  ap2->bitmap[1] = 0;
  ap2->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);
  netconv_p4_action_parameter(ap2);

  send_p4_msg(c, buffer, 2048);
}

void fill_sendout_table(uint8_t port, uint8_t smac[6])
{
    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, "sendout");

    exact = add_p4_field_match_exact(te, 2048);
    strcpy(exact->header.name, "standard_metadata.egress_port");
    exact->bitmap[0] = port;
  exact->bitmap[1] = 0;
    exact->length = 2*8+0;

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

  ap = add_p4_action_parameter(h, a, 2048);
  strcpy(ap->name, "smac");
  memcpy(ap->bitmap, smac, 6);
  ap->length = 6*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);
}

uint8_t macs[MAX_MACS][6];
uint8_t portmap[MAX_MACS];
uint8_t ips[MAX_MACS][16];
int mac_count = -1;

int read_macs_and_ports_from_file(char *filename) {
  FILE *f;
  char line[200];
  int values[6];
  int values_ip[16];
  int port;
  int i;

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

  while (fgets(line, sizeof(line), f)) {
    line[strlen(line)-1] = '\0';
    //TODO why %c?
    if (23 == sscanf(line, "%02x%02x:%02x%02x:%02x%02x:%02x%02x:%02x%02x
   :%02x%02x:%02x%02x:%02x%02x %x:%x:%x:%x:%x:%x %d",
            &values_ip[0], &values_ip[1], &values_ip[2], &values_ip[3],
            &values_ip[4], &values_ip[5], &values_ip[6], &values_ip[7],
            &values_ip[8], &values_ip[9], &values_ip[10], &values_ip[11],
            &values_ip[12], &values_ip[13], &values_ip[14], &values_ip[15],
          &values[0], &values[1], &values[2],
          &values[3], &values[4], &values[5], &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 < 16; ++i )
        ips[mac_count][i] = (uint8_t) values_ip[i];
      portmap[mac_count] = (uint8_t) port;
    } else {
      printf("Wrong format error in line %d : %s\n", mac_count+2, line);
      fclose(f);
      return -1;
    }
  }
  fclose(f);
  return 0;
}

void dhf(void* b) {
  printf("Unknown digest received\n");
}

void init() {
  int i;
  uint8_t smac[6] = {0xd0, 0x69, 0x0f, 0xa8, 0x39, 0x90};
  printf("INIT");
  clock_t begin = clock();
  for (i=0;i<=mac_count;++i)
  {
    fill_ipv6_fib_lpm_table(ips[i], portmap[i], macs[i]);

          if(0 == (i%1000)){ printf("%d inside sleep \n", i);sleep(1);;}
    fill_sendout_table(portmap[i], smac);
          usleep(1000);
  }

  clock_t end = clock();
  double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

      printf ("ctrl Total entries sent %d time %f\n",i,time_spent);
}
```

```

int main(int argc, char* argv[])
{
  uint8_t ip[16] = {0x20,0x01,0x0d,0xb8,0x85,0xa3,0x08,0xd3,0x13,0x19,0x8a
   ,0x2e,0x03,0x70,0x73,0x34};
    uint8_t mac[6] = {0xa0, 0x36, 0x9f, 0x3e, 0x94, 0xea};
  uint8_t port = 1;

  uint8_t ip2[16] = {0x13,0x19,0x8a,0x2e,0x03,0x70,0x73,0x35,0x20,0x01,0x0d
   ,0xb8,0xff,0xff,0x08,0xd3};
    uint8_t mac2[6] = {0xa0, 0x36, 0x9f, 0x3e, 0x94, 0xe8};
  uint8_t port2 = 0;

    uint8_t smac[6] = {0xd0, 0x69, 0x0f, 0xa8, 0x39, 0x90};

  if (argc>1) {
    if (argc!=2) {
      printf("Too many arguments...\nUsage: %s <filename(optional)>\n",
   argv[0]);
      return -1;
    }
    printf("Command line argument is present...\nLoading configuration data
   ...\n");
    if (read_macs_and_ports_from_file(argv[1])<0) {
      printf("File cannot be opened...\n");
      return -1;
    }
  }

  printf("Create and configure l3 test controller...\n");
  c = create_controller_with_init(11111, 3, dhf, init);

  printf("Launching controller's main loop...\n");
  execute_controller(c);

  printf("Destroy controller\n");
  destroy_controller(c);

  return 0;
}
```

Listing C.2 – MACSAD IPv6 Controller code

ANNEX D – P4 Code

D.1 IPv4 Code

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

header_type ipv4_t {
  fields {
    versionIhl : 8;
    diffserv : 8;
    totalLen : 16;
    identification : 16;
    fragOffset : 16;
    ttl : 8;
    protocol : 8;
    hdrChecksum : 16;
    srcAddr : 32;
    dstAddr: 32;
  }
}

parser start {
  return parse_ethernet;
}

#define ETHERTYPE_IPV4 0x0800

header ethernet_t ethernet;

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

header ipv4_t ipv4;
```

```

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

action on_miss() {
}

action fib_hit_nexthop(dmac, port) {
  modify_field(ethernet.dstAddr, dmac);
  modify_field(standard_metadata.egress_port, port);
  add_to_field(ipv4.ttl, -1);
}

table ipv4_fib_lpm {
  reads {
    ipv4.dstAddr : lpm;
  }
  actions {
    fib_hit_nexthop;
    on_miss;
  }
  size : 512;
}

action rewrite_src_mac(smac) {
  modify_field(ethernet.srcAddr, smac);
}

table sendout {
  reads {
    standard_metadata.egress_port : exact;
  }
  actions {
    on_miss;
    rewrite_src_mac;
  }
  size : 512;
}

control ingress {
/* fib lookup, set dst mac and standard_metadata.egress_port */
  apply(ipv4_fib_lpm);

/* set smac from standard_metadata.egress_port */
  apply(sendout);
}

control egress {
}
```

Listing D.1 – IPv4 P4 code

D.2 IPv6 Code

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

header_type ipv6_t {
    fields {
        version : 4;
        trafficClass : 8;
        flowLabel : 20;
        payloadLen : 16;
        nextHdr : 8;
        hopLimit : 8;
        srcAddr : 128;
        dstAddr : 128;
    }
}

parser start {
  return parse_ethernet;
}

#define ETHERTYPE_IPV6 0x86DD

header ethernet_t ethernet;

parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
    ETHERTYPE_IPV6 : parse_ipv6;
    default: ingress;
  }
}

header ipv6_t ipv6;

```

```
parser parse_ipv6 {
  extract(ipv6);
  return ingress;
}

action on_miss() {
}

action fib_hit_nexthop(dmac, port) {
  modify_field(ethernet.dstAddr, dmac);
  modify_field(standard_metadata.egress_port, port);
  add_to_field(ipv6.hopLimit, -1);
}

table ipv6_fib_lpm {
  reads {
    ipv6.dstAddr : lpm;
  }
  actions {
    fib_hit_nexthop;
    on_miss;
  }
  size : 512;
}

action rewrite_src_mac(smac) {
  modify_field(ethernet.srcAddr, smac);
}

table sendout {
  reads {
    standard_metadata.egress_port : exact;
  }
  actions {
    on_miss;
    rewrite_src_mac;
  }
  size : 512;
}

control ingress {
/* fib lookup, set dst mac and standard_metadata.egress_port */
  apply(ipv6_fib_lpm);

/* set smac from standard_metadata.egress_port */
  apply(sendout);
}
```

```

control egress {
}
```

Listing D.2 – IPv6 P4 code