UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Daniel Lazkani Feferman

Design, Implementation and Evaluation of a VXLAN-capable Data Center Gateway using P4

Projeto, implementação e avaliação de um Data Center Gateway compatível com VXLAN usando P4

Campinas
2019

Daniel Lazkani Feferman

Design, Implementation and Evaluation of a VXLAN-capable Data Center Gateway using P4

Projeto, implementação e avaliação de um Data Center Gateway compatível com VXLAN usando P4

Dissertation presented to the Faculty of Electrical Engineering and Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Master in Electrical Engineering, in the area of Computer Engineering

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

Orientador: Prof. Dr. Christian Esteve Rothenberg

Este exemplar corresponde à versão final da dissertação defendida pelo aluno Daniel Lazkani Feferman, e orientada pelo Prof. Dr. Christian Esteve Rothernberg

Campinas
2019

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

F321p

Feferman, Daniel Lazkani, 1992-
Design, implementation and evaluation of a VXLAN-capable data center gateway using P4 / Daniel Lazkani Feferman. – Campinas, SP : [s.n.], 2019.

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 de um data center gateway compatível com VXLAN usando P4
Palavras-chave em inglês:
Software defined networking
Network performance
Computer network performance evaluation
Routing
Área de concentração: Engenharia de Computação
Titulação: Mestre em Engenharia Elétrica
Banca examinadora:
Christian Rodolfo Esteve Rothenberg [Orientador]
Rodolfo Villaça
Marcos Rogerio Salvador
Data de defesa: 22-05-2019
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0002-6481-5116
- Currículo Lattes do autor: http://lattes.cnpq.br/3911976164716041

COMISSÃO JULGADORA - DISSERTAÇÃO DE MESTRADO

Candidato: Daniel Lazkani Feferman RA: 192714
Data da Defesa: 22 de Maio de 2019
Título da Tese: "Design, Implementation and Evaluation of a Data Center Gateway compatible with VXLAN using P4"

Prof. Dr. Christian Rodolfo Esteve Rothenberg (FEEC/UNICAMP)(Presidente)
Prof. Dr. Rodolfo da Silva Villaca (UFES)
Prof. Dr. Marcos Rogerio Salvador (UNICAMP)

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

To all my family and friends

Acknowledgement

God gives us three counselors on our short journey through life: parents, professors, and friends. Parents have been and will always be guiding us. The professors orient us through our technical and professional development. Friends complement the small gaps left behind both of them. So, in the following lines I thank all of them:

To my parents Flavio, Elizabete, and Marcel who influenced in so many different ways, by introducing me the day-to-day learning, providing everything needed in my life and also their persistence on something they did not have the opportunity to have, a high degree. More specifically, I thank my father, for showing me that in some moments of despair, the solution may be right in front of us. My mother, for constantly redefining my concept of perseverance, proving that there is no limit to dreams, that they can always be achieved, no matter the size of the challenge.

To my professor Dr. Christian Esteve Rothenberg for allowing me to learn by being around of some of the smartest people in Brazil and for accepting this huge challenge of orienting me, without professional networking experience, through this such challenging and agile field of computer networks. To my friend and almost a second advisor, Dr. Gyanesh Patra, for the patience of answering most of my questions and increasing my knowledge of innumerable subjects.

To my friends for the advises and technical support. Special thanks to my love and best friend, my girlfriend Natalie for the support through this journey. Since I had a job over the week and the thesis over the weekend, she abdicated most of her weekends to keep helping me to achieve this goal.

I want to express my gratitude for every one of Tim Brazil, who has strengthened some topics in this work.I thank the financial and technical support received from Ericsson Hungary, Silicon Valley, and Brazil. Lastly, I thank the Funcamp for the process nº 35789-17 and 78064/2018. In summary, the phrase “If I have seen further it is by standing on the shoulders of Giants.” of Sir Isaac Newton never made so much sense.

You can't connect the dots looking forward; you can only connect them looking backwards. So you have to trust that the dots will somehow connect in your future. You have to trust in something — your gut, destiny, life, karma, whatever.

Steve Jobs

Abstract

For some years, Software-Defined Networking (SDN) has been revolutionizing the networking landscape, giving administrator users the possibility to program the network control plane. However, the deployment of SDN solutions gave researchers space to new challenges, aiming to upgrade our networks to new levels through deeper data plane programmability.

The Programming Protocol-Independent Packet Processors (P4) is a Domain Specific Language (DSL) to express how packets are processed on a programmable network platform. Considering the objective to allow P4 programmability with high performance, the Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD) uses the OpenDataPlane (ODP) Open Source project to provide specific Application Programming Interfaces (APIs), enabling the interoperability between different hardwares and minimizing the overhead. The MACSAD is a compiler that takes advantage of the P4 language simplicity and ODP APIs flexibility to work on different platforms, but still maintaining high performance. Thus, MACSAD can be called as a ”unified compiler system with high performance”, considering that it can execute the same P4 program on multiple targets with high throughput.

This project aims to add Virtual eXtensible Local Area Network (VXLAN) support to MACSAD, integrate it with an SDN controller, evaluate the throughput, latency and the Load balance distribution through multiple polynomials. Thus, to achieve this integration we will make a P4 VXLAN implementation and an SDN approach to populate the tables through a simple controller.

Finally, we will analyze different load balancing polynomials, mainly through Checksum and CRC functions and a performance evaluation of the whole system, to perform the last one we will take advantage of Network Function Performance Analyzer (NFPA) and Open Source Network Tester (OSNT), generating different types of traffic to benchmark our P4-defined dataplane application.

Key-words: P4, SDN, CRC, Load Balancing, VXLAN, ODP, MACSAD, DSL, OSNT, NFPA, and Computer Networks

Resumo

Por muitos anos as Redes Definidas por Software (SDN) têm revolucionado o comportamento das redes de computadores, dando aos administradores das mesmas a possibilidade de programar o plano de controle da rede. No entanto, a implantação de soluções SDN deu aos pesquisadores espaço para novos desafios, com o objetivo de atualizar nossas redes por meio de uma programação mais detalhada do plano de dados.

O P4 é uma Linguagem de Domínio Específico (DSL) para expressar como os pacotes são processados em uma plataforma de rede programável. Considerando o objetivo de permitir a programação P4 com alto desempenho, o Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD) utiliza o projeto open source OpenDataPlane (ODP) para fornecer APIs específicas, permitindo a interoperabilidade entre hardwares diferentes e minimizando a sobrecarga dos mesmos.

O MACSAD é um compilador que aproveita a simplicidade da linguagem P4 e a flexibilidade das APIs do ODP para trabalhar em diferentes plataformas, mantendo o alto desempenho. Assim, o MACSAD pode ser chamado de um ”sistema de compilador unificado de alto desempenho”, considerando que ele pode executar o mesmo programa P4 em múltiplos *hardwares* com alta performance.

Este projeto tem como objetivo adicionar suporte VXLAN ao MACSAD, integrá-lo a um controlador SDN, fazer uma análise de throughput, latência e da distribuição do balanceador de carga através de múltiplos polinômios. Assim, para alcançar essa integração, faremos uma implementação P4 VXLAN com uma abordagem SDN para preencher as tabelas através de um controlador simples.

Por fim, faremos uma análise dos métodos de balanceamento de carga, principalmente através de funções Checksum e CRC para uma avaliação de desempenho de todo o sistema. Dessa forma, utilizaremos o Network Function Performance Analyzer (NFPA) e Open Source Network Tester (OSNT) para efetuar os testes de throughput e latência, gerando diferentes tipos de tráfego para análise de performance de nosso programa P4 definido na aplicação do plano de dados.

Palavras-chave: P4, SDN, CRC, Balanceamento de Carga, VXLAN, ODP, MACSAD, DSL, NFPA, OSNT e Redes de Computadores

List of figures

1.1 DCG develop process. 19

2.1 The abstract forwarding model 24
2.2 The ODP Architecture. 25
2.3 Comparison of an atchitecture with & without DPDK 25
2.4 The MACSAD architecture 26
2.5 P4 compilation process. 27
2.6 The VXLAN header. 28
2.7 The Network Function Performance Analyzer architecture 29

3.1 DCG use case representation. 34
3.2 DCG pipeline architecture implementation using P4. 35
3.3 use case validation test. 38
3.4 Inbound functional evaluation. 38
3.5 Outbound functional evaluation. 38
3.6 Scenario 1 PCAP files Load Balanced. 40

4.1 The testbeds environments. 44
4.2 Boxplot for latency representation. 45
4.3 Impact of the number of FIB sizes in the Latency for DCG Inbound. 46
4.4 Impact of the number of FIB sizes in the Latency for DCG Outbound. 47
4.5 Throughput with the increase of cores with (256 bytes and 100 entries). 49
4.6 Inbound and Outbound throughput comparison (4 cores and 100 tables entries). 49
4.7 perf cache miss percentage per use cases and driver I/O. 51
4.8 Impact of FIB sizes in the Throughput for DCG with Socket-mmap (four cores experiment). 52
4.9 Impact of FIB sizes in the Throughput for DCG with DPDK (four cores experiment). 52
4.10 Multi-core performance evaluation of Cavium Thunder X on Inbound use case . 54
4.11 Multi-core performance evaluation of Cavium Thunder X on Outbound use case 55
4.12 The Load Balancing extended evaluation script. 57
4.13 IPv4 95 percentile load balancing analysis 58
4.14 IPv4 95 percentile load balancing analysis 59

C.1 The P4 parser representation of the VXLAN program. 75
C.2 The P4 tables dependencies representation of the VXLAN program. 76

E.1 Scenario 2 PCAP files Load Balanced. 84
E.2 Scenario 3 PCAP files Load Balanced. 85
E.2 IPv4 95 percentile of Mean Square Error for different polynomials 87
E.3 IPv4 0x8d95 load balancing analysis 88
E.4 IPv4 0x973afb51 load balancing analysis 89
E.5 IPv4 0xd175 load balancing analysis 90
E.6 IPv4 CRC8 load balancing analysis 91
E.7 IPv4 CRC16 load balancing analysis 92
E.8 IPv4 CRC32 load balancing analysis 93
E.9 IPv4 CRC32c load balancing analysis 94
E.10 IPv6 95 percentile of Mean Square Error for different polynomials 95
E.11 IPv6 CRC8 load balancing analysis 96
E.12 IPv6 CRC16 load balancing analysis 97
E.13 IPv6 CRC32 load balancing analysis 98
E.14 IPv6 CRC32c load balancing analysis 99
E.15 IPv6 0xd175 load balancing analysis 100

List of tables

2.1 ODP supported platforms. 24
2.2 Comparison of the main programmable VXLAN switches. 32

3.1 DCG complexity table 37

Acronyms

ARP	Address Resolution Protocol
API	Application Program Interface
BIDIR-PIM	Bidirectional Protocol Independent Multicast
CLI	Command Line Interface
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DApp	Dataplane Application
DCG	Data Center Gateway
DPDK	Data Plane Development Kit
DSL	Domain Specific Language
DUT	Device Under Test
GCC	GNU Compiler Collection
GENEVE	Generic Network Virtualization Encapsulation
HLIR	High-Level Intermediate Representation
HW	Hardware
IB	Inbound
IR	Intermediate Representation
IGMP	Internet Group Management Protocol
IP	Internet Protocol
IPG	Inter-Packet Gap
LLVM	Low Level Virtual Machine
LPM	Lowest Prefix Match
MAC	Media Access Control
MacS	MACSAD Switch
MACSAD	Multi-Architecture Compiler System for Abstract Dataplanes
NETCONF	Network Configuration
NFPA	Network Function Performance Analyzer
NFV	Network Functions Virtualization
NI	Network Interface
NRMSE	Normalized Root Mean Square Error
NVGRE	Network Virtualization using Generic Routing Encapsulation
OB	Outbound
ODP	OpenDataPlane
OF	OpenFlow
OSNT	Open Source Network Tester

OVS	Open vSwitch
P4	Programming Protocol-Independent Packet Processors
RFC	Request for Comments
RMSE	Root Mean Square Error
RSS	Receive Side Scaling
SDK	Software Development Kit
SDN	Software Defined Networking
SoC	System-on-a-chip
SR-IOV	Single Root I/O Vir-tualization
SW	Software
STT	Stateless Transport Tunneling
TCP	Transmission Control Protocol
VMs	Virtual Machines
VLAN	Virtual Local Area Network
VNI	VXLAN Network Identifier
VTEPs	VXLAN Tunnel End Points
VXLAN	Virtual eXtensible Local Area Network
YANG	Yet Another Next Generation

Summary

1 Introduction **17**

1.1 Thesis Objectives 18

1.2 Methodology 18

1.3 Text Organization 20

2 Background and Literature Review **21**

2.1 Background 21

2.1.1 Software Defined Networking (SDN) 21

2.1.2 Programming Protocol-Independent Packet Processors (P4) 22

2.1.3 OpenDataPlane (ODP) 24

2.1.4 Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD) 25

2.1.5 Virtual eXtensible Local Area Network (VXLAN) 27

2.1.6 Network Function Performance Analyzer (NFPA) 28

2.1.7 Open Source Network Tester (OSNT) 29

2.2 Related work 29

3 VXLAN-based Data Center Gateway Implementation with P4 **33**

3.1 Use Case and Architecture 33

3.2 Prototype implementation 35

3.3 Use case complexity 36

3.4 Functional Validation 37

3.4.1 PCAP analysis 37

3.4.2 Load balancing evaluation 37

3.5 Concluding remarks 41

4 Experimental evaluation **42**

4.1 Methodology 42

4.2 Latency measurements 44

4.2.1 Results Discussion 44

4.3 Throughput 48

4.3.1 Multi-core 48

4.3.2 Scalability 52
4.3.3 Multi-architecture 53
4.4 Extended evaluation on load balancing performance 56

5 Conclusion and future work 60

References 62

A Publications 66

B The DCG P4 code 67

C P4 graphs 75

D The Load Balancing test code 77

E The LB analysis 83
E.1 Functional evaluation 83
E.2 Automated LB analysis 86
E.2.1 IPv4 86
E.2.2 IPv6 95

Chapter 1

Introduction

Considering the exponential growth in packets transmissions over the network, we need to reevaluate how traffic is managed and improved by adding new protocols and functionalities. One of the most sought features by network administrators and the academia is the ability to reconfigure and redesign our networks, or in other words, to give programmability to our systems.

In the past decades, the network architecture had the control and forwarding planes coupled together. Over the years, computer networks have been getting complicated and hard to manage, with routers, switches, firewalls, Network Address Translators, etc (Feamster, Rexford & Zegura 2014). Initially, each vendor implemented the control plane with proprietary solutions, and to configure network device it was necessary to use configuration interfaces that vary across vendors and sometimes even across products from the same vendors. Considering this scenario, Software-Defined Networking (SDN) (Kreutz, Ramos, Verissimo, Rothenberg, Azodolmolky & Uhlig 2014) was born to split both planes, giving the capability to a single software control program to manage multiple data planes from different vendors, two of the current most famous controllers are OpenDaylight (Medved, Varga, Tkacik & Gray 2014) and ONOS (Berde, Gerola, Hart, Higuchi, Kobayashi, Koide & Lantz 2014). The first and most renowned standard interface solution was OpenFlow (McKeown, Anderson, Balakrishnan, Parulkar, Peterson, Rexford, Shenker & Turner 2008), enabling direct access to the control and forwarding layer on devices such as switches and routers.

Though OpenFlow being initially a vast technological advancement, it has limitations such as each new headers need to be implemented on a new version, which can take years to be released. Furthermore, each new version needs to have retro compatibility, making the deployment of it even harder. Ideally, we should be able to give the network precisely which types of headers we want to implement and how they will be parsed, or in other words, to allow our data plane to be programmable. The Programming Protocol-Independent Packet Processors (P4) language (Bosshart, Daly, Izzard, McKeown, Rexford, Schlesinger, Talayco, Vahdat, Varghese & Walker 2013) aims to solve enable a standardized language to enable data plane programmability.

The P4 is an open source project that aims to define how packets are processed; it uses the Match+Action model and can be developed using SDN solutions. The language has three primary goals: **reconfigurability**, meaning that over time we can reconfigure how packets are processed; **protocol independence**, the network administrator can implement or even create new protocols; and **target independence**, details of the switch do not need to be known (Bosshart et al. 2013). Another useful tool in computer networks is the OpenDataPlane (ODP)[1], which is an open source project to enable APIs to develop the data plane.

Combining the simplicity of the P4 language with the flexibility, performance, and portability of ODP APIs, the Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD) (Patra & Rothenberg 2016), (Patra, Rothenberg & Pongracz 2017) was built. The MACSAD compiler converts a P4 program to a Intermediate Representation (IR) and then to C language.

In this work, we present a VXLAN architecture and a P4 solution to this scenario, enabling the division of our networks into multiple virtual networks. However, to successfully achieve this goal, we aim to show that while giving programmability to our VXLAN switch, by using MACSAD we are not compromising features of our network, e.g., increasing latency or decreasing throughput. Then, since our VXLAN architecture features a load balancer through polynomials, we analyze different functions using a new metric to get the best distribution of our load balance applied to the network architecture.

1.1 Thesis Objectives

We can define the main goal of this dissertation as follows: to design, implement and evaluate a VXLAN program using MACSAD compiler and a simple SDN controller. To this end, we identified the following objectives:

- **Design the architecture and implementat a VXLAN-based Data Center Gateway (DCG) pipeline.** To write in P4 the use case pipeline and test it with MACSAD. However, to archive this objective, new primitives need to be added to MACSAD.

- **Performance evaluation.** Given the target DCG implementation, we measure the obtained datapath performance in terms of throughput and latency on different servers and scenarios using state-of-the art traffic generators (NFPA and OSNT).

- **Evaluate the most commons polynomials applicable for load balancing.** By means of adequate metrics, we evaluate different algorithms supported in SW and HW dataplane devices to distribute packet flows over different network paths and leverage the result in our DCG implementation.

1.2 Methodology

In order to achieve our objectives, we found four steps. Figure 1.1 summarizes the main activities in our methodology flow; each step contains sub-steps that may be done in parallel:

[1] https://www.opendataplane.org

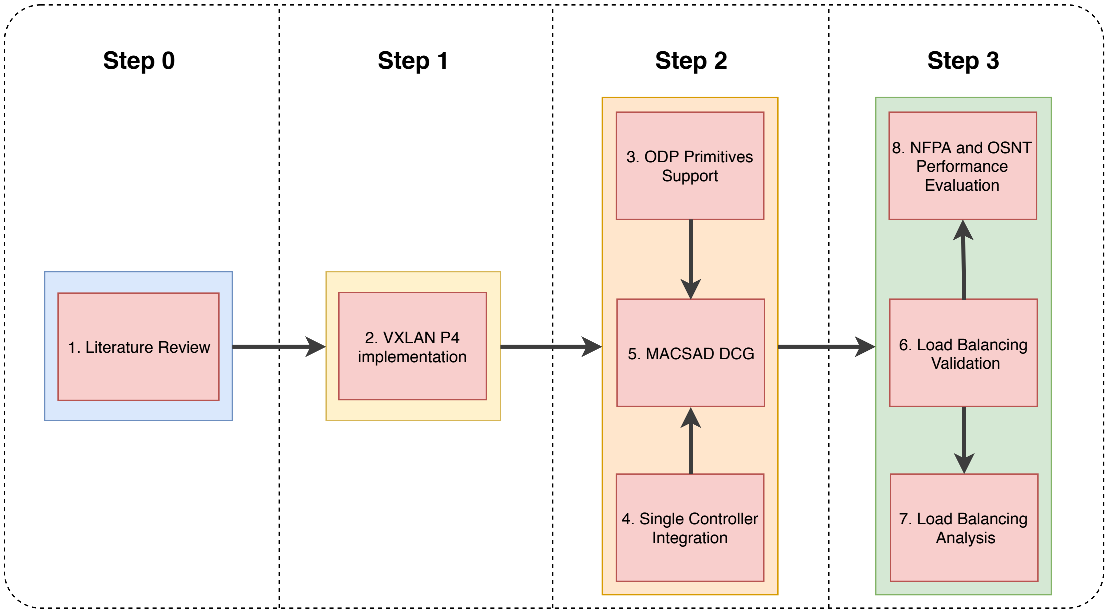

Step 0
Step 1
Step 2
Step 3
1. Literature Review
2. VXLAN P4 implementation
3. ODP Primitives Support
5. MACSAD DCG
4. Single Controller Integration
8. NFPA and OSNT Performance Evaluation
6. Load Balancing Validation
7. Load Balancing Analysis

Figure 1.1: DCG develop process.

1. **Literature review:** as part of any project, we started with the study and analysis of the state of the art of P4 language, load balance polynomials, VXLAN, SDN controller, and MACSAD compiler considering its architecture and support.

2. **VXLAN P4 implementation:** after the literature review, as a second step we have implemented the VXLAN P4 program, which was tested using behavioral-model.

3. **ODP Primitives support:** since MACSAD did not support all the required P4 and ODP primitives, step 3 was the feature implementation on MACSAD using the ODP APIs.

4. **Add support to a simple controller:** in this part we have populated the VXLAN tables through a customized SDN controller, which was managing the packet flow.

5. **MACSAD DCG:** in this sub-step we adapt parts of the executed VXLAN P4 implementation to MACSAD by operating in an emulated environment, meaning the results should be considerably better than a physical (and more realistic) test. In this part, we were able to evaluate the functionality of our DCG P4 program.

6. **Load Balancing Validation:** using the MACSAD DCG we were able to validate the load balance feature and compare it using two different function: CRC32 and checksum16.

7. **Load Balancing Analysis:** through a new metric we have evaluated some of the most commons algorithms to search the best polynomial to load balance traffic over multiple hosts and servers.

8. **NFPA and OSNT Performance Evaluation:** considering the whole network assembled with a P4 VXLAN implementation, MACSAD compatibility and integration with an

SDN controller we have executed the performance tests with different packets I/Os and configurations.

1.3 Text Organization

The remainder of this work is structured as follows. In Chapter 2, we provide background and related works. In Chapter 3, we describe our use case, the VXLAN implementation, and a load balance comparison between CRC and checksum algorithms. In Chapter 4, we evaluate the performance (throughput and latency) of our Data Center Gateway (DCG) P4 program and an extended analysis of the load balancing feature. Lastly, In Chapter 5, we present the conclusion of this work and future work. In Appendix A, we expose the author publications related to this work. In Appendix C we present the parser and dependency table graph representation. In Appendix B we expose the DCG P4 code. In Appendix D, we present the load balance code used to analyze different polynomial. Lastly, in Appendix E, we expose the load balance analysis results into heatmaps.

Chapter 2

Background and Literature Review

In this chapter, we review the literature and industry advancement relevant to our research, along with other proposed solutions.

2.1 Background

This section aims to define the basic concepts of our research. We start by covering the controller that will manage multiple data-planes. Aiming to measure our performance we will execute the Network Function Performance Analyzer (NFPA) (Csikor, Szalay, Sonkoly & Toka 2015) and Open Source Network Tester (OSNT) (Antichi, Shahbaz, Geng, Zilberman, Covington, Bruyere, McKeown, Feamster, Felderman, Blott, Moore & Owezarski 2014), which converts the tests into statistics of throughput (Gbps) and latency (ηs) of the network performance. Finally, the Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD) converts the P4 program to low-level hardware instructions and the VXLAN program, solving our proposed use case scenario.

2.1.1 Software Defined Networking (SDN)

In 2008, UC Berkeley and Stanford University proposed (McKeown et al. 2008) to decouple the network control from the packet forwarding, enabling the control plane to be easily programmed and allowing the network intelligence to be centralized in SDN controllers. Therefore, this revolution on the network architecture led to the development of multiple controllers: Beacon, Floodlight, NOX, POX, Ryu, ONOS and ultimately OpenDaylight.

The SDN controller can be compared to the brain of the network. It acts as the strategic control point to better manage the flow control of the switches and routers to deploy intelligent systems. Thus, the controller is similar to the network core; It lies between network devices at one end and applications at the other end. Any communications between applications and devices need to pass through the controller (Feamster et al. 2014).

2.1.2 Programming Protocol-Independent Packet Processors (P4)

Considering the development of OpenFlow (OF) protocol over the past years, few limitations were found (e.g., most switches have multiple policies and stages of match+action tables, and limited TCAM space). Furthermore, to include a new header on OF it was necessary to update its version with retro-compatibility, making the release of new versions too long (Bosshart et al. 2013). Initially, OF 1.0 started with 12 fields. In 2015, the last version of OF (version 1.5) was released containing more than 40 fields of headers, even the founders of OF recognizes that one of its main problems is that the interface is getting too heavy.

These limitations led to the necessity of an open source language named as "P4" that enables the following feature:

1. The packet parser is configurable and not tied to a specific header format;

2. The Match+Action table is able to match on all defined field and support multiple tables;

3. The header fields and meta-data packet-processing is able to use primitives like *copy*, *add*, *remove*, *modify*;

The P4 language is a huge revolution in networks as it gives programmability to the data plane. A P4 program is composed of five basic components:

- **Tables:** mechanism to make the packet processing. Inside each table there are fields to be matched and actions to be executed;

- **Actions:** P4 allows the construction of actions using simple protocol-independent primitives;

```
action _nop() {
}

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

table L3{
  reads {
    inner_ipv4.dstAddr : lpm;
  }
  actions {
    nhop;
    _nop;
  }
}
```

Listing 2.1: An example of a Layer 3 table using Lowest Prefix Match (LPM) to match the IPv4 destination address with actions forward to next hop or skipping to the next table.

- **Parser:** analyze the packet headers and sequences of the packet;

```
parser parse_ipv4 {
  extract(ipv4);
  return select(latest.fragOffset, latest.ihl, latest.protocol) {
    IP_PROTOCOLS_IPHL_UDP : parse_udp;
    default: ingress;
  }
}
```

Listing 2.2: An IPv4 parser extracting the IPv4 field and passing to the next parser field/control table: UDP or to ingress (first table to match packet fields).

- **Control:** defines the order of match tables with conditional support (“if” and “else”);

```
control ingress {
  if (routing_metadata.res == BONE){
    apply(ARPselect);
  }
  else if (routing_metadata.res == BTWO){
    apply(ownMAC);
    apply(LBselector);
    apply(vxlan);
    apply(L3);
    apply(sendout);
    if (routing_metadata.aux == BTWO){
      apply(vpop);
    }
  }
}
```

Listing 2.3: The packet will first match its fields to the L3 table and then the “sendout” table.

- **Headers:** specifies fields widths and order;

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

header ipv4_t ipv4;
```

Listing 2.4: Each field of an IPv4 header being declared.

The abstract forwarding model of a P4 program is illustrated in Figure 2.1, it shows how a P4 program allows to express a packet processing pipeline by programming the parser, match+action tables, and then deparser. When a packet arrives, its headers are parsed, passed through the P4 tables and action pipeline before the deparser writes the headers back and sends the modified packet.

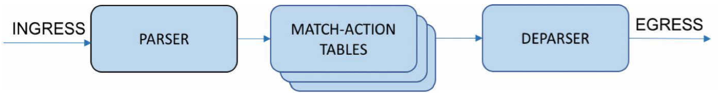


Figure 2.1: The abstract forwarding model. Source: (Patra et al. 2017).

2.1.3 OpenDataPlane (ODP)

The OpenDataPlane (ODP) is an open-source platform that leverage on specific hardware acceleration to support multiple platforms with high performance through a set of APIs for networking data plane. The project supports the following architectures (targets): ARMv7, ARMv8, MIPS64, PowerPC, and x86. In Table 2.1 we expose ODP supported platforms, including manufacturers own implementations.

Table 2.1: ODP supported platforms.

Name	**Owner/Maintainer**	**Target Platform**	**Architecture**
odp-linux	Open contribution maintained by LNG	Any Linux. Support Netmap	Any
odp-dpdk	Open contribution maintained by LNG	Intel x86 using DPDK	Intel x86
odp-keystone2	Texas Instruments	TI Keystone II SoCs	ARM Cortex-A-15
linux-qoriq	NXP	NXP QorIQ SoCs	Power & ARMv8
OCTEON	Cavium Networks	Cavium Octeon^{TM} SoCs	MIPS64
THUNDER	Cavium Networks	Cavium ThunderX^{TM} SoC	ARMv8
Kalray	Kalray	MPPA platform	MPPA
odp-hisilicon	Hisilicon	Hisilicon platform	ARMv8

In Figure 2.2 we expose the ODP stack with the work-flow for an ODP application, which is different from a standard Linux app. An ODP app is linked to one of the ODP implementations of Table 2.1 and optimized to a specific hardware platform (Server or SoC). Then, the Vendor Specific Hardware Blocks and Software Development Kit (SDK) is called to finally gets to the hardware platform. Although this process initially seems complicated, as it has more blocks to be called, the real difference can be seen because of specific optimized hardware functions that allow higher throughput.

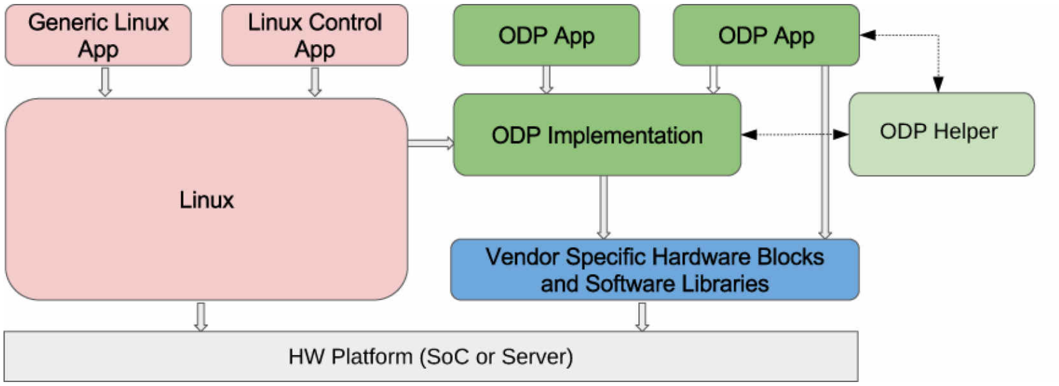

Generic Linux App
Linux Control App
ODP App
ODP App
Linux
ODP Implementation
ODP Helper
Vendor Specific Hardware Blocks and Software Libraries
HW Platform (SoC or Server)

Figure 2.2: The ODP architecture. Source: https://www.opendataplane.org

One of the main highlights of ODP is the possibility to improve performance (throughput and latency) by using specic APIs for the target architecture and the compatibility with Intel DPDK (Pongracz, Molnar & Kis 2013) and Netmap (Rizzo 2012). The DPDK is a Linux Foundation project consisting of specific drivers and libraries to allow Intel's devices to improve its performance by creating a fast packet processing Dataplane Application (DApp). The DPDK started on x86 architectures, and it was later expanded to ARM and IBM Power chips. In Figure 2.3 we compare a group of applications with and without DPDK, as can be noted, the DPDK libraries bypass the network drivers allowing a massive increase of throughput.

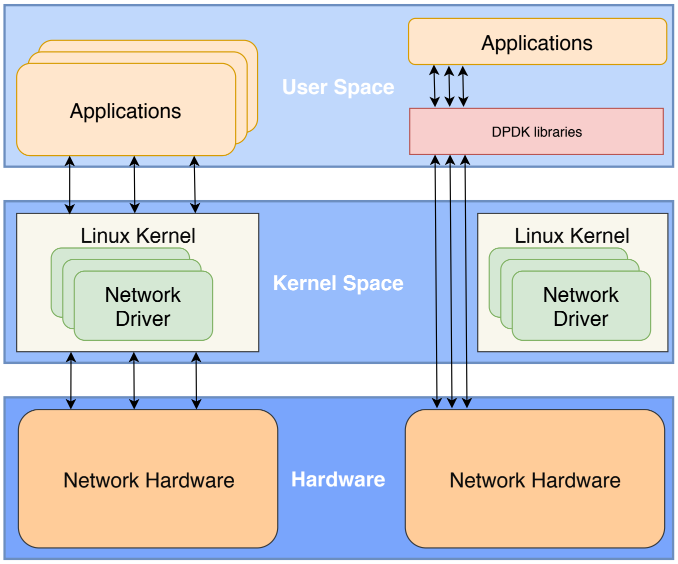


Figure 2.3: Comparison of an atchitecture with & without DPDK

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

The MACSAD is a P4 compiler that focuses on high performance with portability and flexibility. As shown in Figure 2.4, the MACSAD is composed of three main modules:

- **Auxiliary frontend:** in simple words, this module is responsible for several Domain Specific Language (DSL) aggregation. It creates an Intermediate Representation (IR) of the P4 program, which is used by the core compiler. In this module, the P4-hlir project is used to translate P4 programs into a High Level Intermediate Representation (HLIR). The yellow square on Figure 2.5 represents the conversion of a P4 program to a High-Level Intermediate Representation (HLIR).

- **Auxiliary backend:** this module aims to give a standard SDK, using ODP APIs. Furthermore, it contains developed libraries to allow the connection between P4 and ODP.

- **Core compiler:** includes the transpiler and compiler modules. It merges the result of the frontend (the HLIR) and backend (the ODP APIs) to provide the binary which will be used by the device either by a Virtual Machine (x86), Raspberry Pi (ARM), server (x86) or an SoC (ARM).

  The *Transpiler* receives the result from the Auxiliary frontend and automatically generates the Data-path Logic codes. This tool is responsible for the definition of the size, lookup mechanism, and type of tables that will be created using the target's resources. The group of ".c" files generated by the transpiler contains ODP APIs, helper libraries and parts of the P4 program. Furthermore, using this mechanism, we can take advantage of the "Dead Code Elimination" feature, simplifying and optimizing the code using dependency graph of parser logic.

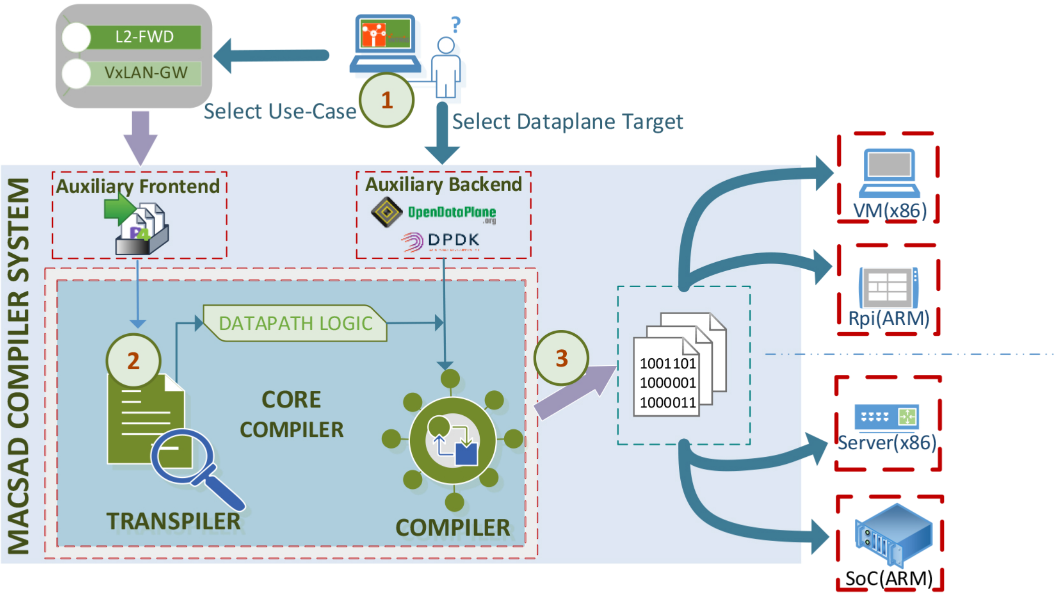


Figure 2.4: The MACSAD architecture (Patra et al. 2017)

The *Compiler* uses the generated *".c"* codes to create the switch for the target; in our project, we will create a VXLAN router using a P4 program. The red squares in Figure 2.5 expose the conversion of the Core Compiler from an HLIR to C files, and then the compiler converts it to a binary representation of the MACSAD Switch (MacS). Currently, MACSAD uses Low Level Virtual Machine (LLVM) and GNU Compiler Collection (GCC) compiler to guarantee the support of multiple targets.

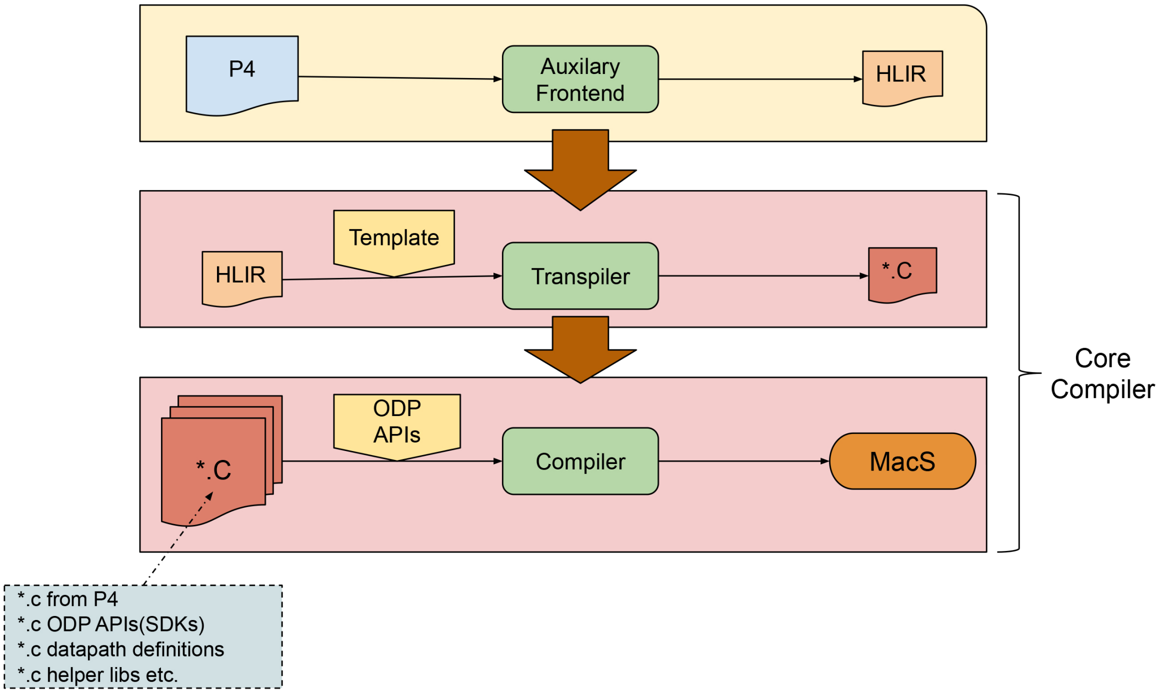


Figure 2.5: P4 compilation process (Patra et al. 2017).

2.1.5 Virtual eXtensible Local Area Network (VXLAN)

Considering the deployment of a massive cloud computing and the usage of server virtualization, the network started to have multiple Virtual Machines (VMs) and each one of them with its Media Access Control (MAC) address. Thus, to ensure the communication with an enormous amount of VMs it was necessary to update huge MAC address tables. Initially, the best solution was to divide the network using the multi-tenancy Virtual Local Area Network (VLAN) protocol. However, this protocol has a limit of only 4,096 VLANs, which can be easily exceeded by today's data centers. Thus, to fulfill this scenario with a vast number of Virtual Machines on an overlay network we need to encapsulate the packet to be sent over a logical "tunnel", which is the Virtual eXtensible Local Area Network (VXLAN) (Mahalingam, Dutt, Duda, Agarwal, Kreeger, Sridhar, Bursell & Wright 2014) protocol, providing scalability with a capacity to support up to 16 million tenants.

The VXLAN protocol is a data plane encapsulation technique aiming to extend the already existing VLAN. The VXLAN solution on data-centers is transparent to the final user since it can only see a regular Internet Protocol (IP) routing flow. In this work, we intend to present a Data Center Gateway architecture through a VXLAN protocol solution, allowing millions of different Virtualized Machines in the network to work without independently assigning MAC address conflicts. Another common problem in large data centers is the overflow problem, where the switch stop learning new addresses until idle entries age out. This scenario causes flooding with an unknown destination. Through VXLAN protocol we intend to better address this problem by taking advantage of the VXLAN Tunnel End Points (VTEPs), dividing the table load and considerably decreasing the chances of this issue. Furthermore, using VXLAN protocol with Bidirectional Protocol Independent Multicast (BIDIR-PIM), we can achieve multicast by mapping VXLAN VNI and multicast IP groups. Then, the VTEPs can provide Internet Group Management Protocol (IGMP) membership reports to the upstream switch/router to join/leave the VXLAN-related IP multicast groups as needed. Lastly, the proposed DCG use case can easily exceed the 4,096 VLAN limit. Thus, the VXLAN limit of millions of different VXLAN Network Identifier (VNIs) proves to be a necessity in this architecture. In Figure 3.1 is shown a packet structured with VXLAN, highlighting some of the most critical bytes of each header (including the VXLAN).

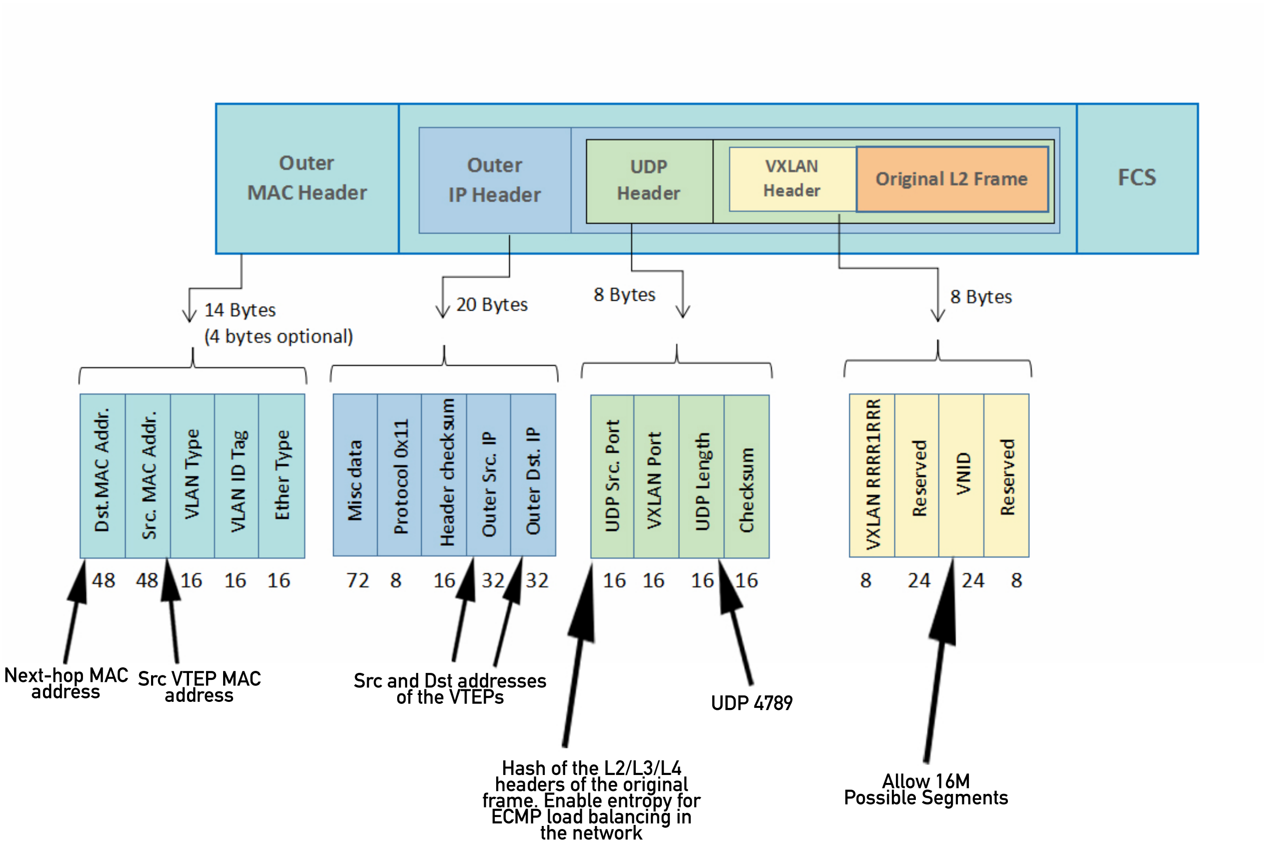


Figure 2.6: The VXLAN header. Adapted from: https://community.fs.com/blog/qinq-vs-vlan-vs-vxlan.html

2.1.6 Network Function Performance Analyzer (NFPA)

The Network Function Performance Analyzer (NFPA) (Csikor et al. 2015) is a benchmarking tool that allows users to measure the performance of network functions by combining software and hardware. Furthermore, the result of these metrics can be compared to other results in the database. The NFPA follows standardized methodologies based on a specific RFC (Bradner & McQuaid 1999).

The NFPA frontend is implemented using Python language, and it has a configuration file to establish the traffic traces and parameters that will be later used. This tool uses Pkt-

Gen[1] (Turull, Sjödin & Olsson 2016), (Robert Olsson 2005) to avoid kernel performance limitations with network card drivers by taking advantage of Intel's Data Plane Development Kit (DPDK)[2]. One of the most exciting features of this analyzer is the ability to generate Gnuplot graphs[3] based on the performance results and compare it with other Network Functions. In Figure 2.7 we expose the NFPA architecture.

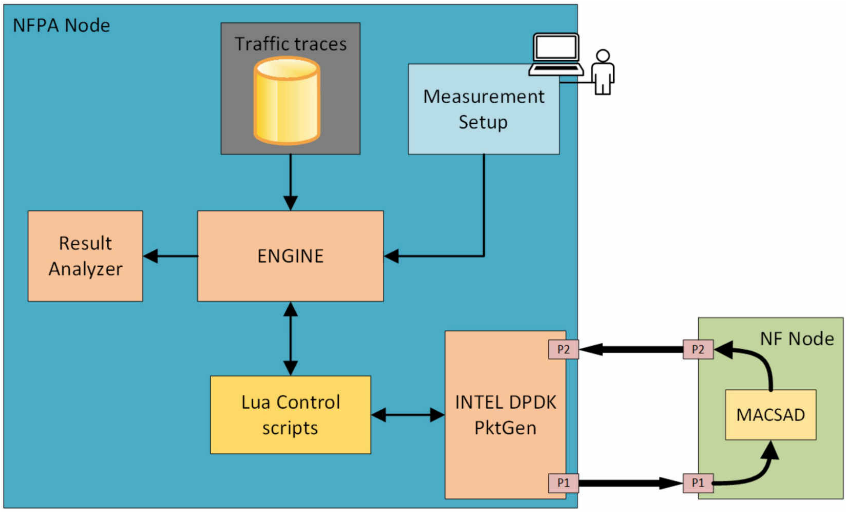


Figure 2.7: The Network Function Performance Analyzer architecture. Source: (Csikor et al. 2015).

2.1.7 Open Source Network Tester (OSNT)

The Open Source Network Tester (OSNT) is an Open Source software for testing network throughput and latency. The testing tool works on top of the NetFPGA platform. The OSNT support NetFPGA-10G and NetFPGA-SUME cards, with full line rate through four 10G Ethernet ports. In this work, we use the NetFPGA-SUME cards donated by the NetFPGA organization. Using a GPS input, the hardware module controls clock drift and phase coordination allowing OSNT to adds 64 bits time-stamps for the latency test with minimal overhead.

The traffic generator uses a PCAP file to send packets. The latency is measured as a per-packet delay time with a high-resolution time-stamp to measure the Device Under Test (DUT). Lastly, to allow this accuracy, the OSNT time-stamp is located right before the transmission of the 10GbE MAC module.

2.2 Related work

There has been a recent interest in Domain Specific Languages (DSL) to achieve a fully programmable network. Currently, the P4C[4] and behavioral-model[5] can be considered the re-

[1] https://pktgen-dpdk.readthedocs.io/
[2] http://dpdk.org
[3] http://www.gnuplot.info
[4] https://github.com/p4lang/p4c
[5] https://github.com/p4lang/behavioral-model

ference on P4 Language, as they both have full compatibility with $P4_{14}$ and $P4_{16}$. The work in Programming Protocol-Independent Packet Processors (Bosshart et al. 2013) introduces the P4 language with its central concepts, including headers, parsers, tables, actions, and control programs. Similarly, packetC (Duncan & Jungck 2009) is a DSL language even more expressive than P4 by allowing access to packet payloads and also stateful processing by providing synchronization constructs for globally shared memory. However, both compilers have the same drawback, they are only used as a reference, and they do not achieve line-rate, for most use cases they can get up to a few Mbps. Protocol-Oblivious Forwarding (Song 2013) share similar goals of P4, but it uses tuples to treats packet headers, the result is a low-level model that resembles the Assembly language. However, while this approach has some undeniable advantages to the compiler side, it does come with the cost of programming packet parsing considerably more complex.

Using P4 language, PISCES (Shahbaz, Choi, Pfaff, Kim, Feamster, McKeown & Rexford 2016) is a compiler that converts P4 programs into a software switch derived from Open vSwitch (OVS)[6], a hardwired hypervisor compatible switch using C. However, PISCES optimize the code in a way that can generate the same OVS switch with much shorter code, up to 40 times shorter. Furthermore, PISCES implementation is protocol independent, supporting new protocols that can be added as new features.

The work in “DC.p4” (Sivaraman, Kim, Krishnamoorthy, Dixit & Budiu 2015) exposes a software Data-Center Switch using P4 that can be compared to a single-chip shared-memory used in many data centers today. Although the article achieves a fully compatible P4 switch with VXLAN protocol, it does not achieve comparable hardware dependent performance, since it uses behavioral-model [7]. Furthermore, commercial products featuring high-performance switches with programmable pipeline include Cisco’s Unified Access Dataplane (Diedricks 2015), Intel’s FlexPipe (*Intel® Ethernet Switch FM6000 Series* 2017) and Cavium’s Xpliant (*Cavium / XPliant® CNX880xx* 2015).

In order to get up to 10Gbps, the work in *“Removing Roadblocks from SDN: OpenFlow Software Switch Performance on Intel DPDK”* (Pongracz et al. 2013), analyze the performance increase of an OpenFlow switch using Intel DPDK. Their software switch supports OpenFlow 1.3 with throughput from 5.26Gbps to 9.60Gbps for packets of 64 and 512 Bytes, respectively. Lagopus (Rahimi, Veeraraghavan, Nakajima, Takahashi, Nakajima, Okamoto & Yamanaka 2016) is another software switch to take advantage of Intel DPDK; it has L2 and L3 functionalities with OpenFlow support. The authors reported a throughput of up to 9.8Gbps, with a packet size of 1500B and 100K entry tables. The Project Translator for P4 switches T4P4S[8] (Voros 2018) is another P4 compiler that takes advantage of DPDK to allow high performance to multiple targets. However, T4P4S uses DPDK as the auxiliary backend, while MACSAD uses DPDK and ODP, enabling the portability of different architectures easier without loosing performance. Similarly, we intend to expose results of Gbps through DPDK using a programmable P4 switch with Load Balance, and VXLAN enabled. Lastly, MACSAD DCG share some of the best features of each of these compilers: the programmability of behavioral-model, the performance

[6] http://openvswitch.org

[7] https://github.com/p4lang/behavioral-model

[8] https://github.com/P4ELTE/t4p4s

of OVS and multi-architecture of T4P4S. Through MACSAD we are able to achieve all this features in a single compiler. In Table 2.2 we present a summary of the main programmable VXLAN software switches.

Table 2.2: Comparison of the main programmable VXLAN switches.

Related Work	P4 Data plane	Target	Platform	Remarks
DC.P4: Programming the Forwarding Plane of a Data-Center Switch	Yes	Software Switch	Behavioral Model or P4C	P4 reference design, full language support (v14 and v16)
OvS	No	Software Switch	OvS	Remote configuration protocol with C and Python bindings
PISCES	Yes	Software Switch	OvS, DPDK	OvS based
T4P4S SMGW	Yes	Multi-Target	T4P4S, DPDK	Support of different platforms
MACSAD DCG	Yes	Multi-Target	MACSAD, ODP, DPDK	Support of different platforms. Portability through ODP APIs and Load Balance function

Chapter 3

VXLAN-based Data Center Gateway Implementation with P4

In this chapter, we will briefly describe the Data Center Gateway architecture and its application using P4 language. Lastly, we will perform a load balancing validation experiment.

3.1 Use Case and Architecture

With the proliferation of cloud computing, an increased number of Virtual Machines (VMs) have been implemented aiming at logical isolation of could applications and tenants. So far, the Virtual LAN (VLAN) protocol has been ubiquitously used to create smaller broadcast domains to substantially decrease the complexity of traffic management among physically not collocated VMs and reduce the cost of broadcast floods. However, due to the limited number of different VLAN ids (4,096) it supports, it has become obsolete as today's data centers need to handle hundreds of thousands of VMs at the same time, and the pace of this increasing numbers is not about to slow down soon. The battle of the network virtualization mechanisms (e.g., NVGRE (Garg & Wang 2015), GENEVE (Sridhar & Wright 2014), STT (Davie & Gross 2016)) is still far from being concluded, and there is no undisputed winner: every vendor tries to push its solution (e.g., Cisco has VXLAN-capable devices, VMware is behind STT), and every solution has its advantages and disadvantages (Pepelnjak 2012). For our proposed design, we chose Virtual eXtensible Local Area Network (VXLAN) (Mahalingam et al. 2014), which supports up to 16M logical networks, and at the same time, is transparent to the endpoints. Virtual Tunnel End Point (VTEP) plays an essential role in the implementation of Virtual eXtensible Local Area Network (VXLAN) with two primary functions: to encapsulate and transport L2 traffic over L3 network, and decapsulate the packets before sending out to the destination.

As sho shown in Figure 3.1, the DCG use case is based on VXLAN tunnels to interconnect different hosts over the Internet redundant servers with the same IP address (*Server 1* and *Server 2* with IP: 8.8.8.1) inside a data center. The VXLAN protocol can serve a multitude of features in data centers using multi-tenancy with different VNI. Basically, the pipeline architecture of a DCG can be divided into two steps:

- **Inbound (IB):** *Host* (with IP: 213.1.1.1) tries to send a packet to a web service identified

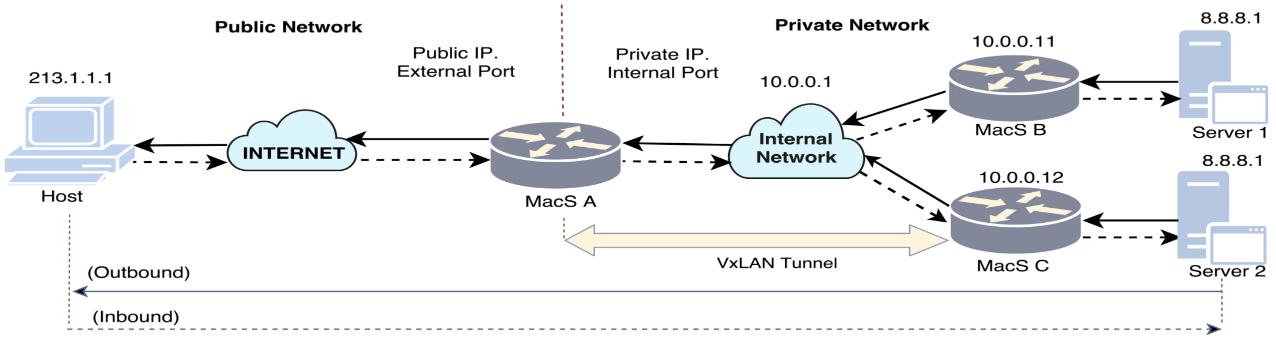


Figure 3.1: DCG use case representation.

by IP address 8.8.8.1 (see Figure 3.1). When the packet reaches the first ingress router *MacS A* (VTEP) of the data center, first a load balancing is carried out (usually based on the source IP address) to determine the next VTEP. Assume that *MacS A* decides to send the packet towards *MacS B*. *MacS A* adds outer L2 (destination MAC of *MacS B*), L3 headers (destination IP set to 10.0.0.11), UDP header and VXLAN header to the packet before sending it out. Finally, *MacS B*, being the second leg of the VXLAN tunnel, decapsulates the packet and send it to *Server 1*.

- **Outbound (OB):** as a response, *Server 1* sends a packet towards *Host* using its original source IP address as destination IP address. When the packet reaches *MacS B*, the packet is encapsulated in a similar way as in the reverse direction, and finally when *MacS A* receives the packet it removes the additional VXLAN headers, rewrites the addresses and send the packet towards *Host* over the Internet.

Load Balancing. One objective of the DCG is to balance the load of the destination servers while avoiding packet disorder. Thus, we opt to use a per-flow load-balancing approach, since per-packet should increase packet disorder (Singh, Chaudhari & Saxena 2012), by using functions over IPs we can guarantee that the same host in normal conditions will be attended by the same server. The functions receive the host source IP, calculates the polynomial result and load balances it by the following function:

$$LB = poli \ \% \ N \tag{3.1}$$

Where:

- **LB:** a number representing a specific server;
- **poli:** the result of a polynomial function (either crc-32, checksum, adler, etc);
- **N:** number of total servers excersing a specified function to be balanced;

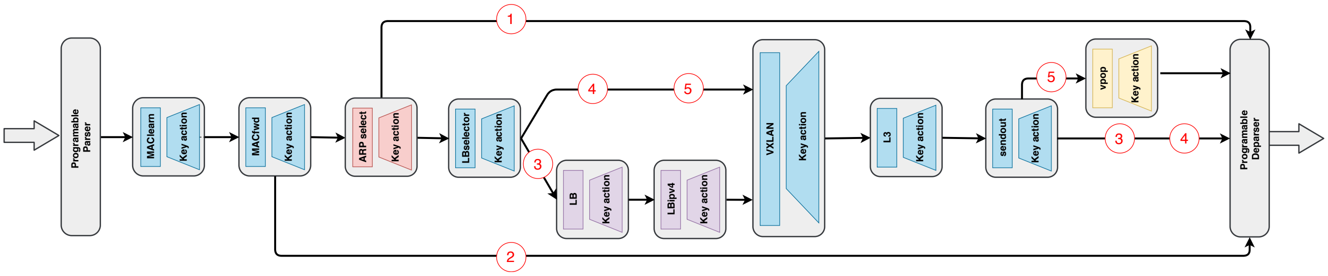


Figure 3.2: DCG pipeline architecture implementation using P4.

3.2 Prototype implementation

Considering the DCG use case presented in Figure 3.1, we implemented the corresponding pipeline in P4 as shown in Figure 3.2. There are three different datapath flows from the parser until the the deparser. The first two (i and ii) represent packets coming from and going to the internal network, while the third one (iii) represents the *Inbound* and *Outbound* use case scenarios:

(i) The first (represented by the red “1”) flow occurs when the switch receives an unknown destination address sent from/to the internal network. Then, an Address Resolution Protocol (ARP) request is made to the control plane in an attempt to find the correct host;

(ii) The second (represented by the red “2”) flow occurs when the switch recognizes with a match a packet from/to the internal network, acting as a simple L2 forward switch;

(iii) For the *Inbound* and *Outbound* use cases represented, the flow starts with 3 (*MacS A* on the *Inbound* and *MacS B* on the *Outbound*), 4 (The internal cloud) and then 5 (*MacS B* on the *Inbound* and *MacS A* on the *Outbound*).

Note that if there is no match on any of the cases above, the packet is dropped, and nothing happens. According to the operation explained above, we generated different VXLAN-encapsulated traffic traces for IB and OB, respectively. The IB traffic includes packets with random source MAC addresses and IPs (to enable Receive Side Scaling (RSS) for our multi-core setup) and the same destination IP set to the server’s IP (8.8.8.1), while the OB has the server’s IP (8.8.8.1) as source IP address and random destination MAC addresses and IPs (again, for enabling RSS in our multi-core setting) simulating various replies to different hosts.

In order to enable more sophisticated and scalable processing, and avoid cross-product problem (Barham, Park, Weatherspoon, Zhou, Chase & Dean 2013) the pipeline consists of multiple matching tables (Open Networking Foundation May 2015), i.e., each flow table has its purpose such as ARP and routing. Altogether, the Inbound have a total of nine matching tables, while the Outbound have eight matching tables. The learning switch table is pre-populated with source IPs and MAC addresses, and the load balancing feature is implemented by a CRC32 function through the source IP address. The VXLAN encapsulation adds the right headers and port numbers prior to MAC address re-writing. Furthermore, for all performance tests we are populating the tables bidirectionally, meaning that the use case being tested contain entries for both use cases, allowing a more realistic scenario.

3.3 Use case complexity

Considering the prototype exposed before, we consider the necessity of a complexity table with parameters that could decrease MACSAD and other compilers performance. Since P4 allows to reprogram the dataplane similar to common languages such as C, Python, and Java allow multiple solutions to the same goal, we expect the same behavior to P4 language. Thus, we consider primordial to present a complexity table that compares the use cases to understand which parameters could decrease the compiler performance. In Table 3.1 we expose the complexity table applied to IB and OB use cases and divided by topics:

- **Parsing:** refers to parse the header and its fields. In the next chapter we expose that this part is specially relevant since the OB packet is considerably heavier than the IB;

- **Processing:** have information of the tables and the pipeline. E.g.: The number of tables each use case need to match;

- **Packet modification:** as the name suggest, the main functions that modify the structure of the header is considered in this part. While the IB copy some headers (encapsulation of Ethernet and IPv4) and add others (VXLAN and UDP headers), the OB remove the headers (Ethernet, IPv4, VXLAN and UDP headers);

- **Metadata:** is local information shared all over the P4 program. E.g.: the IB use this to pass the result of the hash function to another table to perform the load balance;

- **Action complexity:** summarizes the fields and destination expressions modified. Each function that change a field of the packet can be considered in this parameter. Thus, on both use cases there is a modification of the destination address, which would be an example of field writes.

- **Lookups:** Hash (exact) or Lowest Prefix Match (LPM), this parameter is used on every table to match it. While some tables have an exact match, like the ine performing the Load Balance feature, others use LPM, like the table matching the IPv4 destination address.

Table 3.1: DCG complexity table

	Complexity field	**Inbound**	**Outbound**
Parsing	# Packet headers	3	7
	# Packet fields	17	38
	# Branches in parse graph	3	5
Processing	# Tables (no dep)	11	11
	Depth of pipeline	9	8
	Checksum on/off	on	off
	Table size	100	100
State accesses	# Write to different register	0	0
	# Write to same register	0	0
	# Read to different registers	0	0
	# Read to same register	0	0
Packet modification	# Header adds	4	0
	# Header copies	2	0
	# Header removes	0	4
Metadata	# Metadatas	4	3
	Metadata size (bits)	28	12
Action complexity	# Field writes	27	9
	# Arithmetic expressions	2	1
	# Boolean expressions	0	0
	# Externs	1	0
Lookups	# Hash_lookups [key_lenght(bits)]	2[48], 2[16], 2[32], 1[24], 1[9]	2[48], 1[16], 2[32], 1[24], 1[9]
	# LPM [key_length(bits)]	1[32]	1[32]

3.4 Functional Validation

3.4.1 PCAP analysis

In order to validate our P4 model, we have tested it with the P4 reference compiler, the Behavioral model[1]. In Figure 3.3 we expose our validation test, we run a script to build two virtual interfaces (`veth0` and `veth1`) and then we manually populate the tables through a Command Line Interface (CLI). Then, using Scapy([2]) the packets are sent from `veth0` to `veth1`. Finally, using TCPDUMP we save the input and output as a PCAP file. In Figures 3.4 and 3.5 we expose the input and output of the PCAPs using Wireshark for both Inbound and Outbound use cases respectively.

3.4.2 Load balancing evaluation

Through ODP functions we made a few tests exposing the performance of our Load Balance using two different functions: CRC32 and Checksum (16 bits). Thus, we created three PCAPs files containing 1024 random source IPs each and sent it to MACS to Load Balance it. The result of the first PCAP can be seen in Figures 3.6 and the other two can be seen at Appendix E, where the X-axis represents the load balancing metadata used by our DCG P4 program (LB in Equation 3.1), Y-axis represents the number of IPs received by each server, and the vertical line in each bar represents its standard deviation. As can be seen, both functions overload some

[1] Behavioral Model: https://github.com/p4lang/behavioral-model

[2] Scapy: https://github.com/secdev/scapy

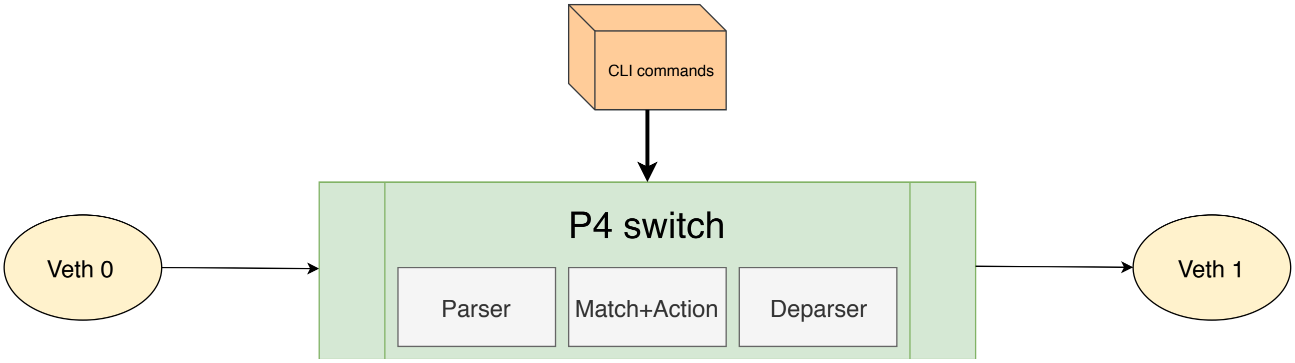
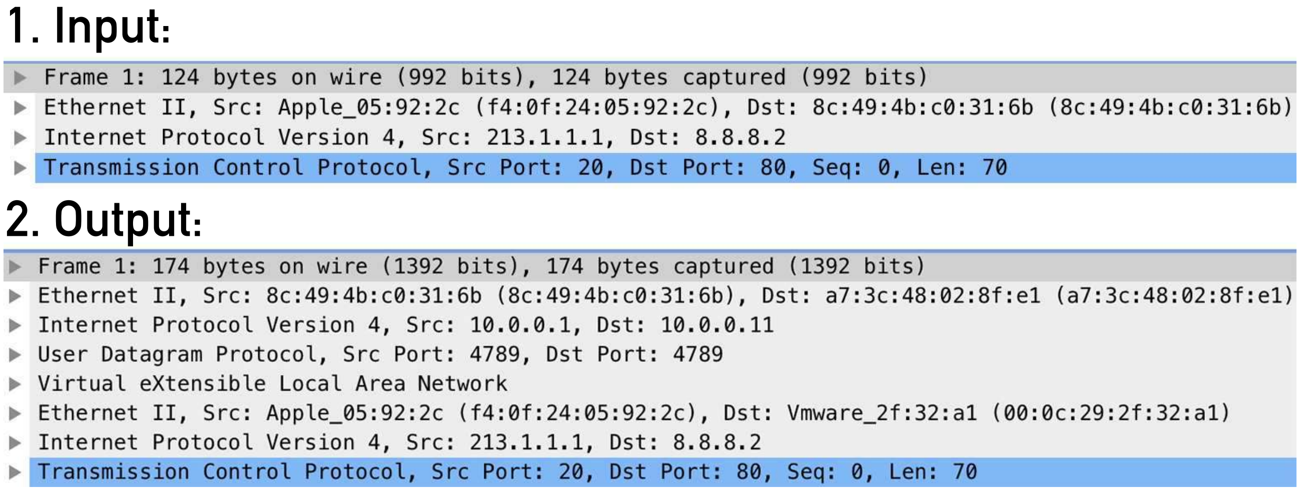


Figure 3.3: use case validation test.



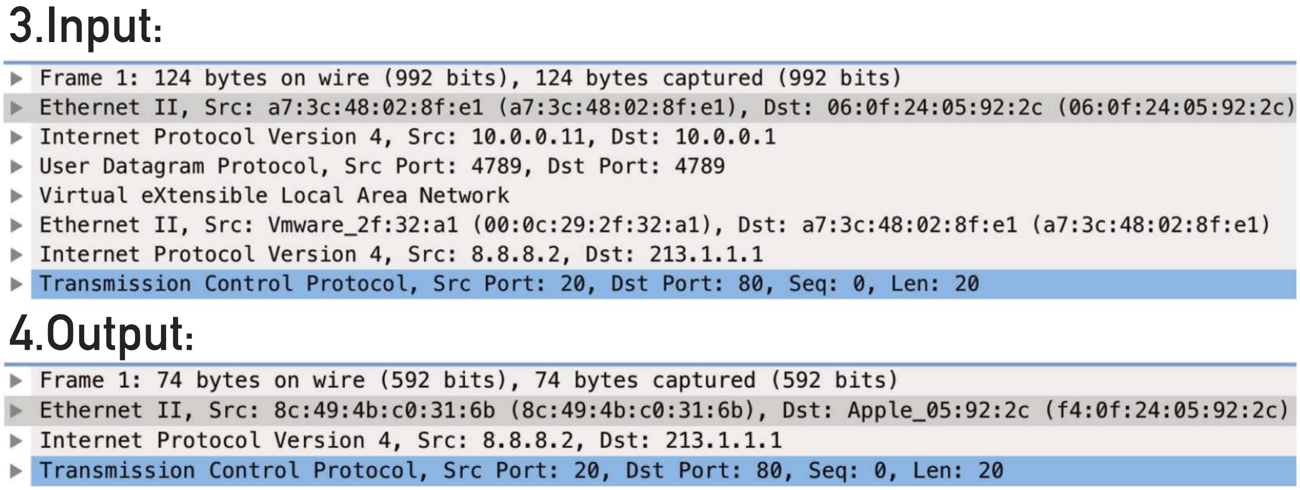
1. Input:

```
▶ Frame 1: 124 bytes on wire (992 bits), 124 bytes captured (992 bits)
▶ Ethernet II, Src: Apple_05:92:2c (f4:0f:24:05:92:2c), Dst: 8c:49:4b:c0:31:6b (8c:49:4b:c0:31:6b)
▶ Internet Protocol Version 4, Src: 213.1.1.1, Dst: 8.8.8.2
▶ Transmission Control Protocol, Src Port: 20, Dst Port: 80, Seq: 0, Len: 70
```

2. Output:

```
▶ Frame 1: 174 bytes on wire (1392 bits), 174 bytes captured (1392 bits)
▶ Ethernet II, Src: 8c:49:4b:c0:31:6b (8c:49:4b:c0:31:6b), Dst: a7:3c:48:02:8f:e1 (a7:3c:48:02:8f:e1)
▶ Internet Protocol Version 4, Src: 10.0.0.1, Dst: 10.0.0.11
▶ User Datagram Protocol, Src Port: 4789, Dst Port: 4789
▶ Virtual eXtensible Local Area Network
▶ Ethernet II, Src: Apple_05:92:2c (f4:0f:24:05:92:2c), Dst: Vmware_2f:32:a1 (00:0c:29:2f:32:a1)
▶ Internet Protocol Version 4, Src: 213.1.1.1, Dst: 8.8.8.2
▶ Transmission Control Protocol, Src Port: 20, Dst Port: 80, Seq: 0, Len: 70
```

Figure 3.4: Inbound functional evaluation.



3.Input:

```
▶ Frame 1: 124 bytes on wire (992 bits), 124 bytes captured (992 bits)
▶ Ethernet II, Src: a7:3c:48:02:8f:e1 (a7:3c:48:02:8f:e1), Dst: 06:0f:24:05:92:2c (06:0f:24:05:92:2c)
▶ Internet Protocol Version 4, Src: 10.0.0.11, Dst: 10.0.0.1
▶ User Datagram Protocol, Src Port: 4789, Dst Port: 4789
▶ Virtual eXtensible Local Area Network
▶ Ethernet II, Src: Vmware_2f:32:a1 (00:0c:29:2f:32:a1), Dst: a7:3c:48:02:8f:e1 (a7:3c:48:02:8f:e1)
▶ Internet Protocol Version 4, Src: 8.8.8.2, Dst: 213.1.1.1
▶ Transmission Control Protocol, Src Port: 20, Dst Port: 80, Seq: 0, Len: 20
```

4.Output:

```
▶ Frame 1: 74 bytes on wire (592 bits), 74 bytes captured (592 bits)
▶ Ethernet II, Src: 8c:49:4b:c0:31:6b (8c:49:4b:c0:31:6b), Dst: Apple_05:92:2c (f4:0f:24:05:92:2c)
▶ Internet Protocol Version 4, Src: 8.8.8.2, Dst: 213.1.1.1
▶ Transmission Control Protocol, Src Port: 20, Dst Port: 80, Seq: 0, Len: 20
```

Figure 3.5: Outbound functional evaluation.

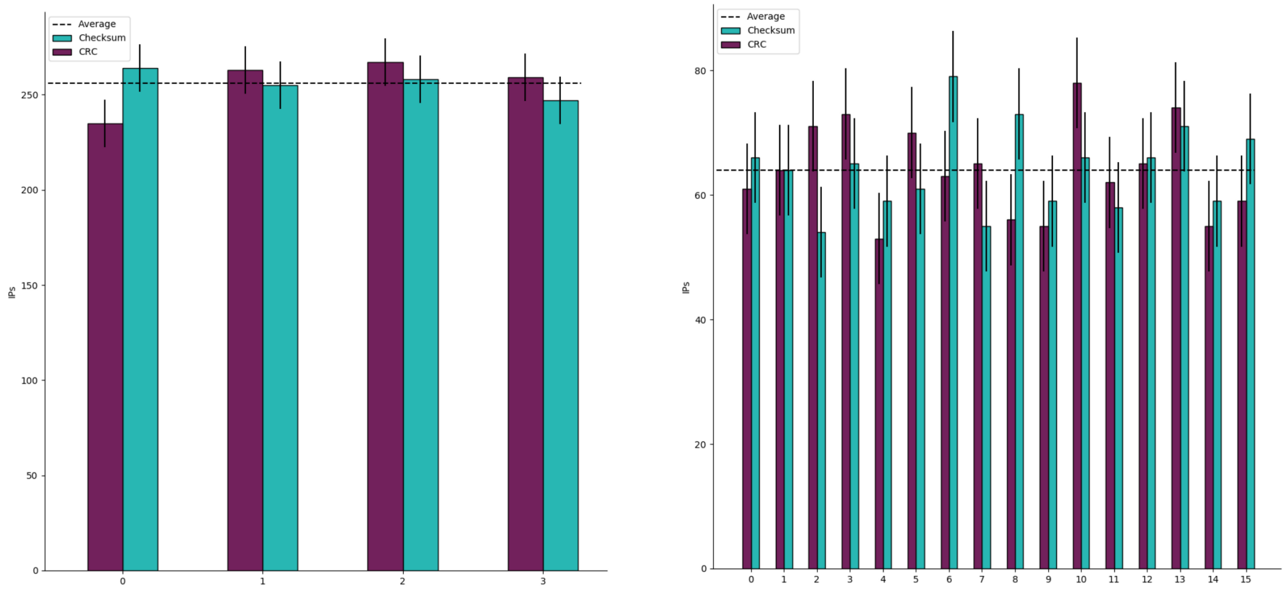
servers while others are under-loaded, which were not intended, since we wanted to distribute equally the traffic. Initially, we wanted to evaluate which of both functions would be nearest to what we considered the optimal distribution, represented in Figures 3.6 as " Average", which is the function:

$$Average = \frac{IP}{n} \tag{3.2}$$

Where:

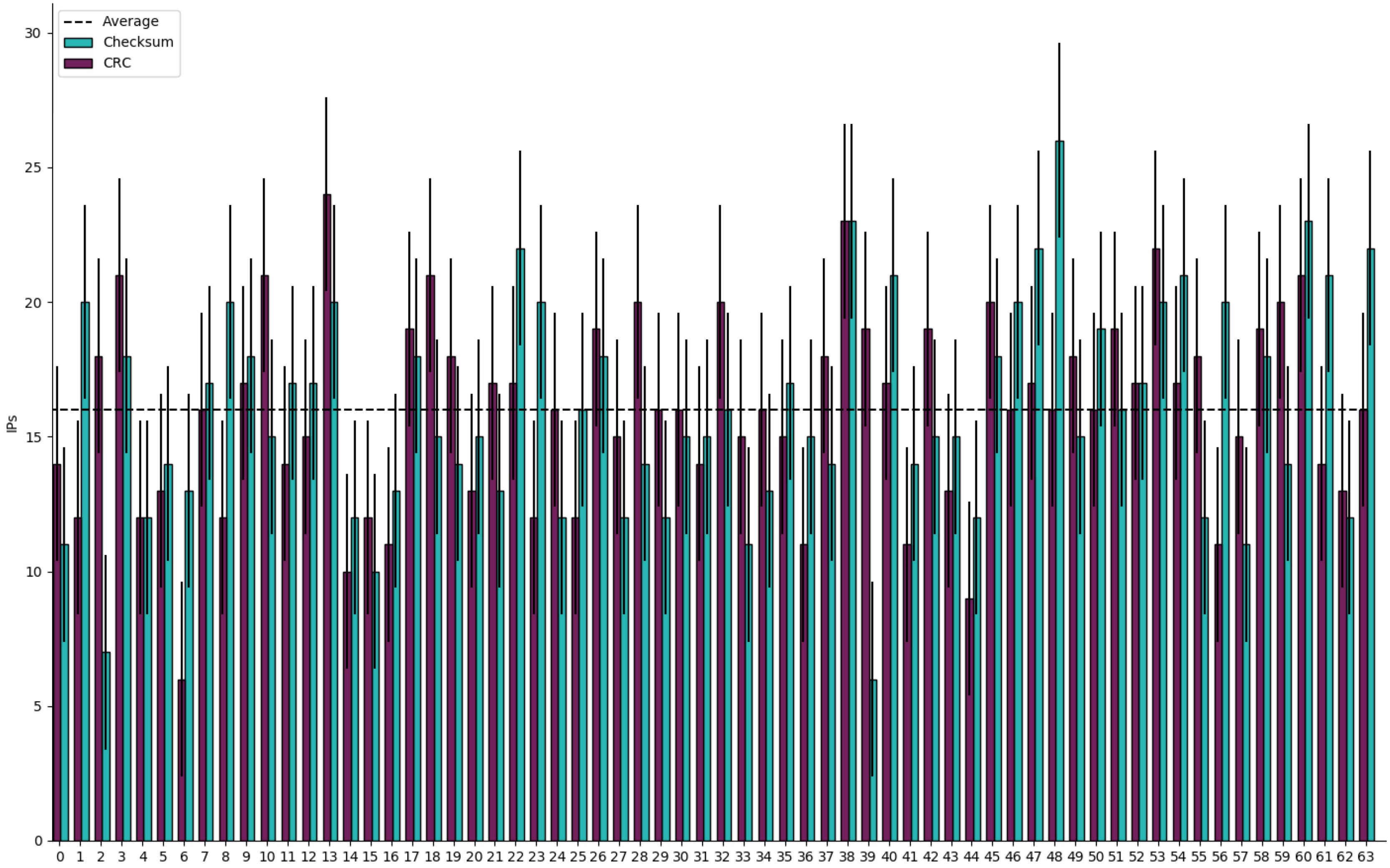
- **IP:** represents the total of IPs sent (1024 IPs);
- **n:** number of servers being balanced;

However, by comparing both functions we were not able to state a clear winner, but since we had similar load balancing performance for both functions and an implemention of checksum by hardware is a costly solution compared to CRC, we opted to discard the use of checksum on our DCG P4 program and use CRC32. Furthermore, in Figure 3.6 (a) we observe the difference of the real distribution versus the optimal distribution is around 10% ($n = 4$), when we increase it to 64 servers as exposed in Figure 3.6 (c) we see that this difference in percentage can get more than 50% in some cases.



(a) Load balance between four servers ($n = 4$)

(b) Load balance between sixteen servers ($n = 16$)



(c) Load balance between sixty-four servers ($n = 64$)

Figure 3.6: Scenario 1 PCAP files Load Balanced.

3.5 Concluding remarks

In this chapter, we have introduced the DCG architecture which allows a large data center with vast number of VMs to attend multiple hosts without issues of MAC conflict and flooding. Furthermore, through P4 language we implented a prototype of these two use scenarios (Inbound and Outbound). Then, we expose the complexity table, which compares both use cases. In subsection 3.4.1 the functional validation of the DCG model is presented, where we send a packet from one interface and check the received packet on the other interface. Finally, in subsection 3.4.2 we validate the load balance feature of our switch using two polynomials: checksum-16 and CRC32.

Chapter 4

Experimental evaluation

In this chapter, we aim to stress our P4-based DCG implementations into three main directions:

- **Performance:** In this analysis we consider the throughput in terms of Gbps and the latency in μs;

- **Scalability:** While a programmable switch with a throughput near the line-rate and latency of a few μ is one of our main goals; we must test the same P4 program with more entries, allowing a much more realistic scenario and exposing that the overhead by scaling through ;

- **Multi-architecture:** we test the throughput of MACS with two architectures, ARM and x86, exposing that the compiler and the DCG P4 program can be run and compared between multiple architectures;

Lastly, we present a novel metric applied to multiple polynomials seeking the best algorithm for a load balancer.

4.1 Methodology

In this section, we introduce our methodology to analyze the throughput and latency over three different servers. In Figures 4.1 (a) and (b) we expose our methodologies. First, we start by measuring the latency and multi-core throughput using OSNT[1]. The OSNT sends packets in one interface, MACS process the packets using the match+action model and sends it to another interface connected to OSNT. In the following text box we present the hardware configuration of the Device Under Test (DUT) for this experiment:

[1] OSNT: http://osnt.org

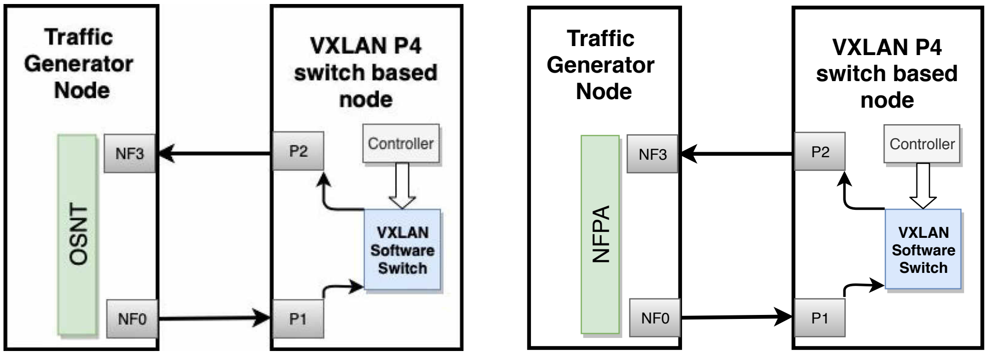
- Processor: Intel Xeon D-1518 processor four cores with two threads per core running at 2.20 Ghz.
- Memory: 32GB*2 DDR4 SDRAM
- Operating System: Elementary OS 0.4.1 Loki (Linux kernel 4.13.0-32-generic)
- NIC: dual-port 10G SFP+
- ODP (v1.19.0.0)
- DPDK (v17.08)

Then, we perform our scalability experiment using NFPA. In this scenario, the NFPA sends packets from one interface using DPDK and receives the result of MacS on the other interface (Bradner & McQuaid 1999). With the testbed configuration, packet loss only occurs when the DUT becomes a physical bottleneck, and therefore the packet rate received by NFPA is representative of the raw performance. This test was conducted according to the following testbed:

- Processor: Intel Xeon E5-2620v2 processor six cores with two threads per core running at 2.00 GHz.
- Memory: 8GB*4 DDR3 SDRAM
- Operating System: Ubuntu 16 LTS (Kernel 4.4)
- ODP (v1.16.0.0)
- DPDK (v17.08)

Lastly, we analyze the multi-architecture capability of MacS by running it on a Cavium Thunder X. Once again we use NFPA to analyze the throughput of the following testbed:

- Processor: 48*2 cores ARMv8
- Memory: 16GB*8 DDR4
- Operating System: Ubuntu 18.04 LTS
- odp-hunderx (v1.11.0.0)
- DPDK (v17.08)



(a) The multi-core testbed.

(b) The scalability and multi-architecture testbed.

Figure 4.1: The testbeds environments.

4.2 Latency measurements

The OSNT time-stamp the packets transmitted and received in 32 bits values each, under a pre-configured position. Unfortunately, the latency test of OSNT was explicitly made to test fixed packets size with 64 bits free for the time-stamp to be written, which is a problem for small packets sizes and use cases that change the position of a header over the packet, since we add and remove headers on our P4 program, we have faced this issue.

In this work, we present an approach to achieve the latency measurement by taking advantage of P4 dataplane programmability. We have modified our P4 program to parse the time-stamp arriving at the MacS. Then, we copy its header to a medatada and remove the packet without loosing the time-stamp of the OSNT transmission step. Lastly, we add the time-stamp back to the original position of the packet, allowing OSNT to read the time-stamp and compare with the received time. Although this approach will not allow us to precisely measure the latency of our program, since it should add some overhead to our DCG program, we will be able to state that our P4 program achieves less than the measured latency for each driver and packet size.

4.2.1 Results Discussion

The DCG latency experiments are tested by sequentially sending 55 packets repeated 100 times and IPG set in 100000 to bring a stable conditions to measure the latency experiments. The OSNT manage the traffic rate by configuring the Inter-Packet Gap (IPG). Our method follows others work such as (Kawashima, Nakayama & Hayashi 2017) to bring a stable condition to measure our latency experiments. Since we need to add 16 Bytes for the timestamp, we were not able to test it with 64 Bytes (Inbound) and 110 Bytes (Outbound). Figure 4.2 represents a sample of the statistics parameters (99% outliers, median, mean, etc) analyzed in Figures 4.3 and 4.4. As exposed in Figures 4.3 and 4.4 there is a clear relation between the packet size and the observed latency, where an increase in the first one result to an increase to the last one. Furthermore, we observed that as expected the latency of socket-mmap is considerably higher than DPDK, for both use cases we found the socket-mmap has a latency of at least two times higher than DPDK. In Figure 4.3 the maximum latency is observed for the Inbound using DPDK with 10K entries (26μ seconds) and Socket with 100 entries (39μ seconds). In Figure 4.4 the maximum latency is observed for the Inbound using DPDK with 100K entries (10μ seconds) and Socket with 100K entries (30μ seconds).

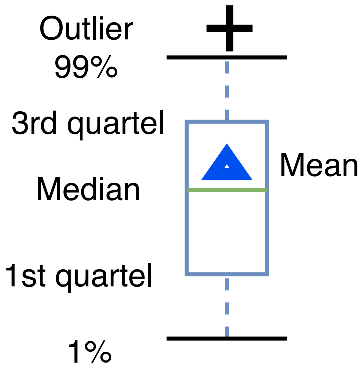


Figure 4.2: Boxplot for latency representation.

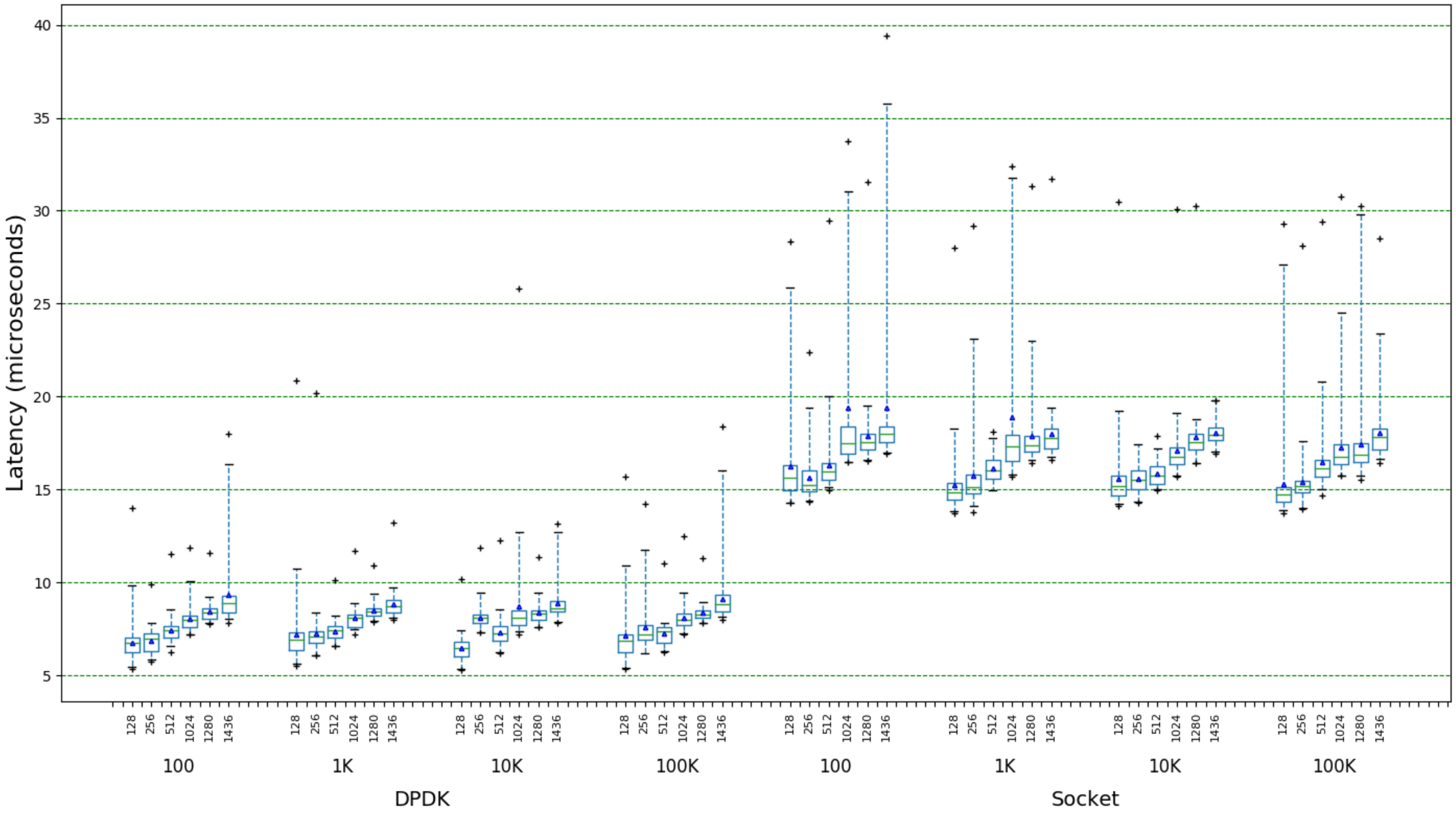


Figure 4.3: Impact of the number of FIB sizes in the Latency for DCG Inbound.

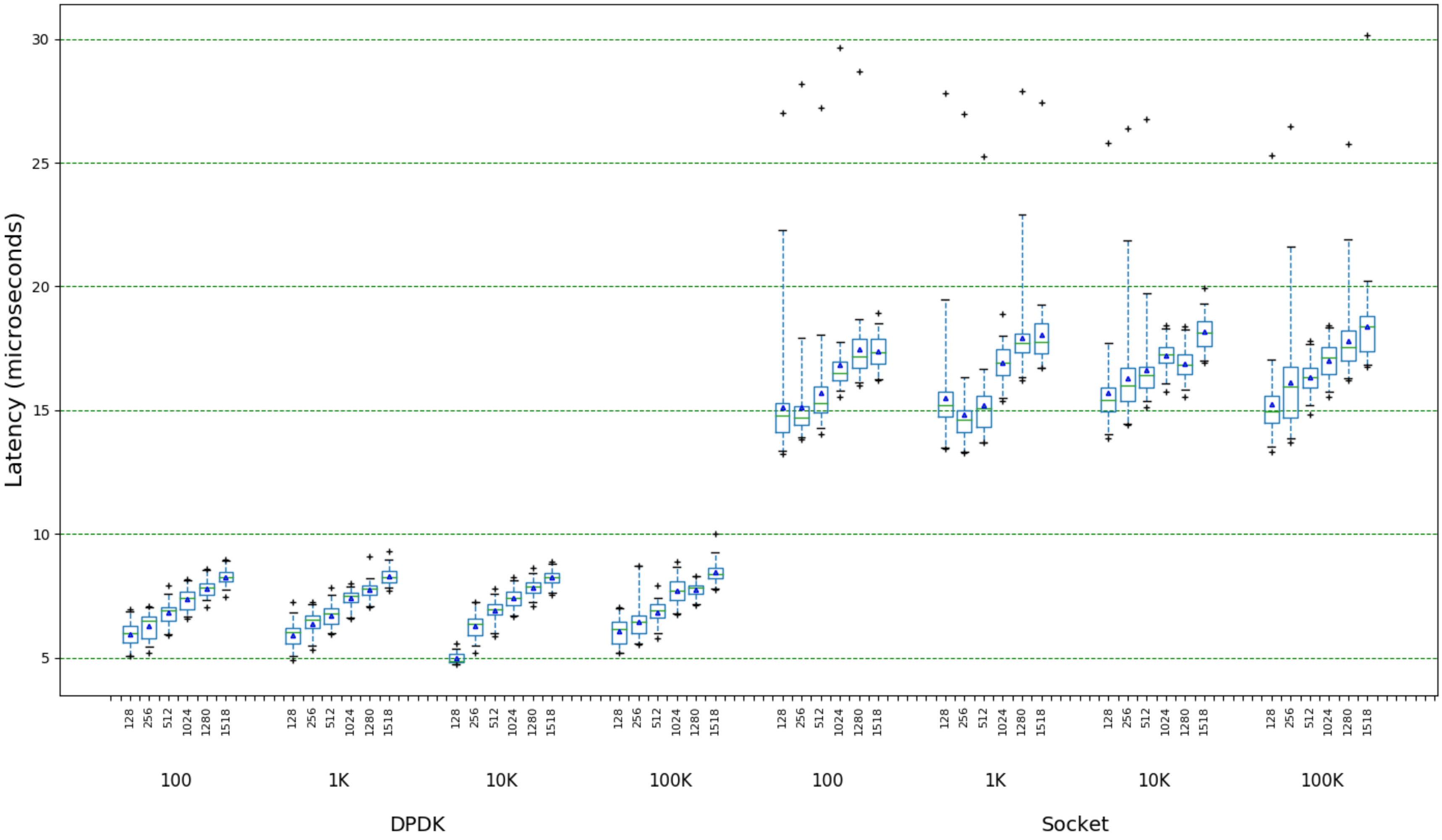


Figure 4.4: Impact of the number of FIB sizes in the Latency for DCG Outbound.

4.3 Throughput

In this section, we analyze the throughput of our DCG on three different targets: two Xeon servers based on x86 architecture and Cavium Thunder X using ARMv8. The following experiments use NFPA and OSNT to analyze the performance based on three aspects:

- The throughput while increasing the number of cores. In this test, we compare the multi-threading feature of MACS and analyze the cache-misses for each packet I/O and use case. This test is conducted using OSNT;
- The performance in Gbps of while increasing the entries using NFPA with up to 100k entries; This test is conducted using NFPA;
- The throughput of an energy efficient server using ARMv8 architecture with up to 96 cores. This experiment is conducted using NFPA;

All throughput measurements were conducted for at least 60 sec (Bradner & McQuaid 1999), and every data point in our performance measurements is an average value. Confidence intervals are unnecessary as results are stable and reproducible for all frameworks.

4.3.1 Multi-core

Considering that we are building switches inside servers with multiple cores, it is expected that more cores will increase the throughput. This experiment was conducted on the same DUT of section 4.2.1 using the topology of Figure 4.1 (a). Thus, we ran MACSAD from two to six cores increasing by steps of two. Figure 4.5 expose the results for both: Socket-mmap and DPDK drivers. As can be seen, there is a boost in performance as the number of physical cores increases. However, we observe that by increasing the number of threads with more than the limit of physical cores (by using hyperthreading), the throughput decreases since MacS does not allow execution units to stay idle during a clock cycle and bus bandwidth limitation (Schöne, Hackenberg & Molka 2012).

The result in term of throughput (Gbps) is shown in Figure 4.6, where the left-hand side corresponds to the Inbound use case, while on the right-hand side, the results for the Outbound use case are depicted in function of increasing packet sizes; note that for the Outbound the 64 bytes were switched by the additional 50 bytes overhead imposed by the VXLAN headers and 6 bytes of data.

There is a clear relation between the packet size and the throughput achieved, this is due to a restriction on the bus-bar interruption, by increasing this interruption limit (e.g., by adding Mellanox[2] network card), we would achieve similar throughput to all packet sizes, which would be near the line rate for DPDK and almost 8 Gbps for Linux default drive (Socket-mmap). Besides the noticeable performance improvement with the increasing number of packet sizes, we can observe a somewhat counter-intuitive performance difference between the two use cases: VXLAN encapsulation in the middle of the pipeline for IB sub-case refreshes cache which is

[2] http://www.mellanox.com/

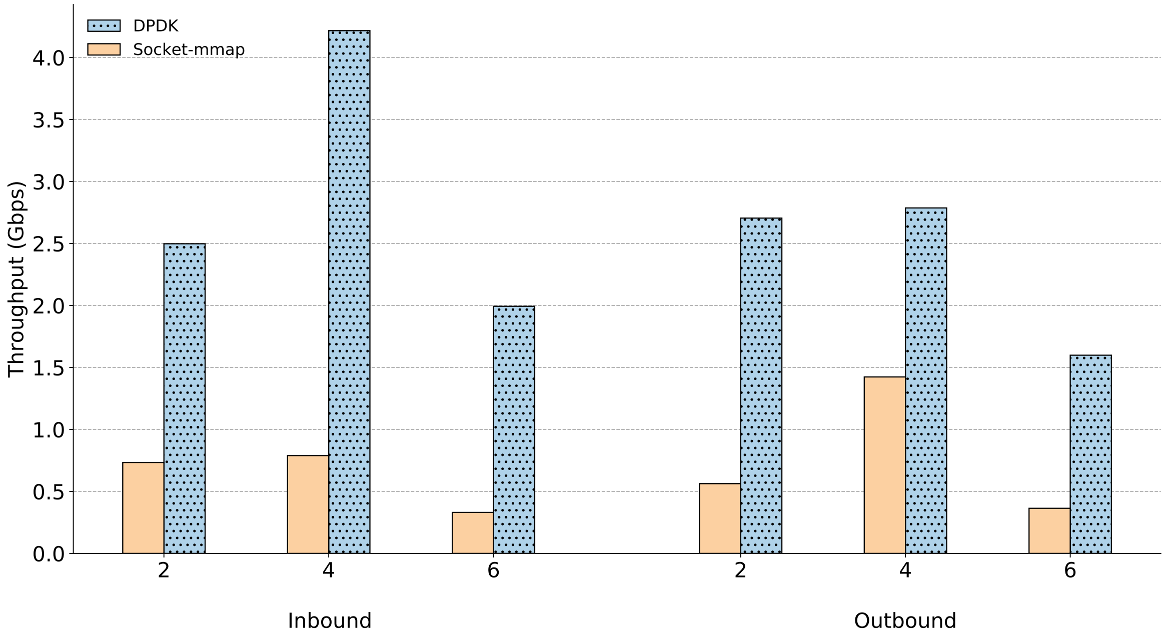


Figure 4.5: Throughput with the increase of cores with (256 bytes and 100 entries).

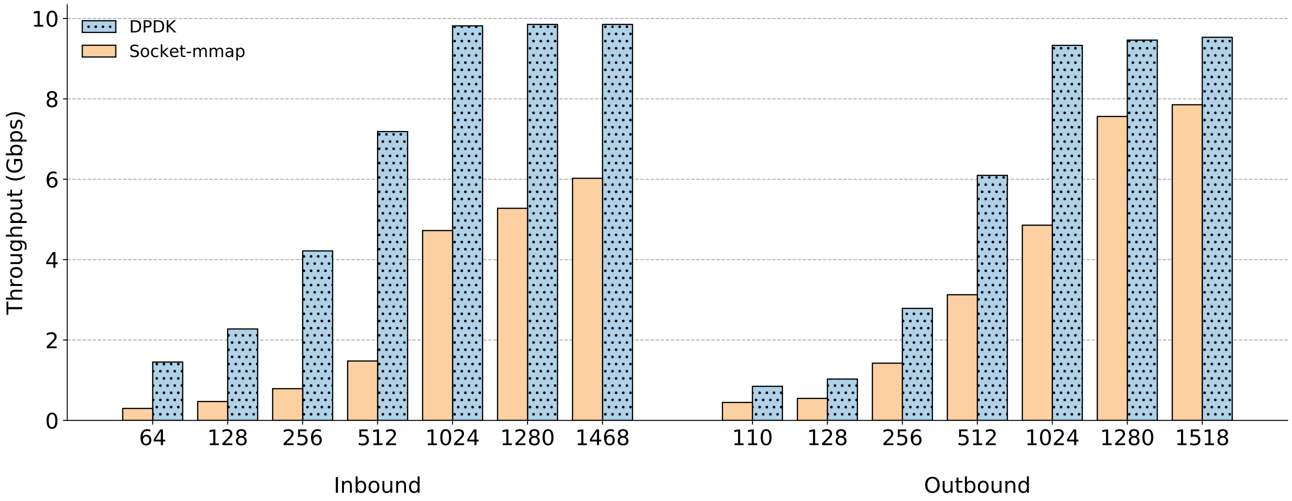
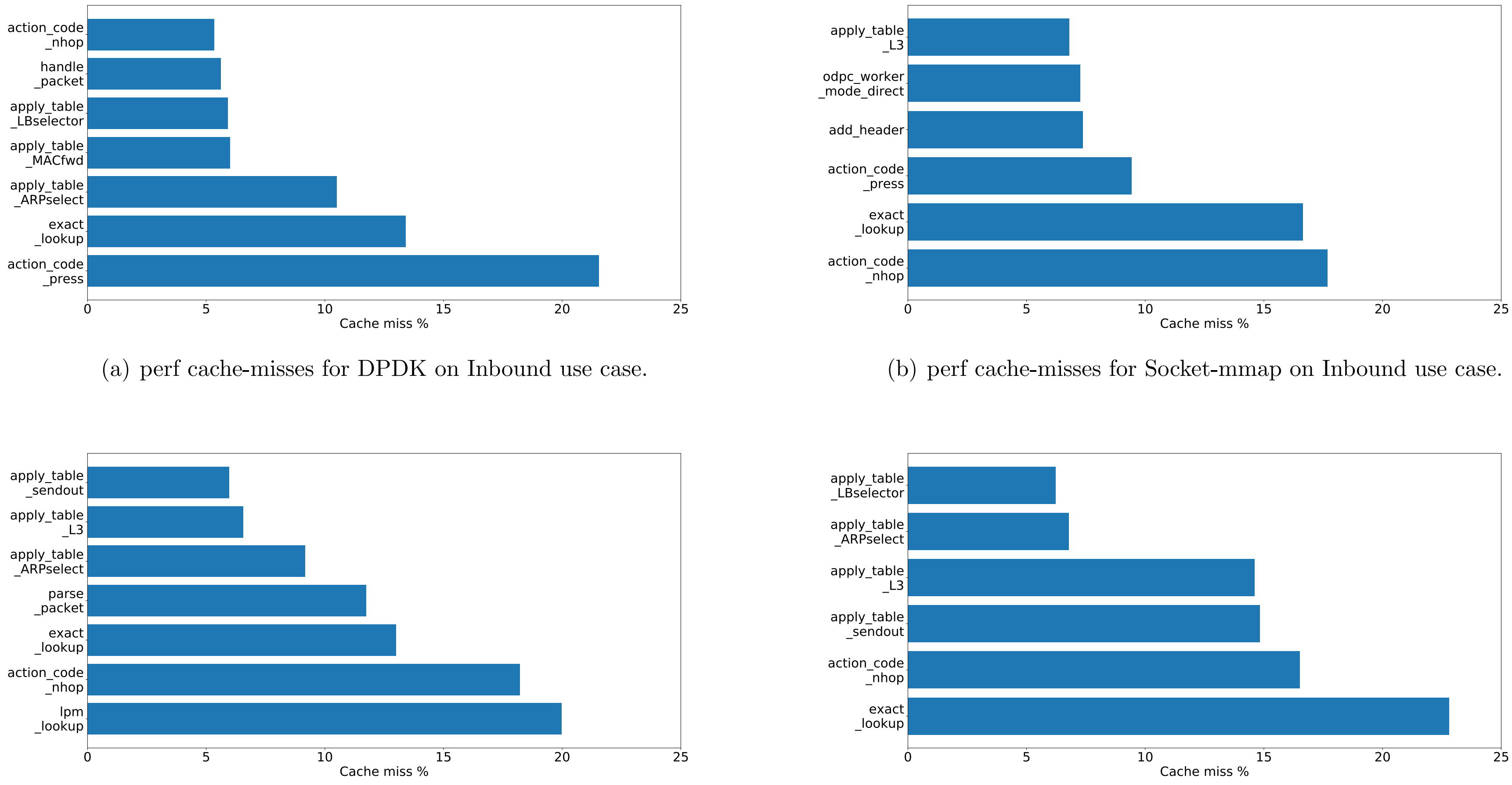


Figure 4.6: Inbound and Outbound throughput comparison (4 cores and 100 tables entries).

leveraged by tables further down in the pipeline, whereas decapsulation happens at the end of the pipeline for OB resulting in higher cache miss. Furthermore, the OB packet contains 7 headers to parse, while the the IB has just 3, meaning that MacS need to parse much more headers and fields on the OB, decreasing the performanc.

Considering the throughputs obtained in Figures 4.7 we seek to take a closer look at what is stressing the processor. The perf command (de Melo 2010) is a powerful tool that allows users to count the number of events (e.g., cache-misses and instructions executed). Using perf we evaluate the CPU cycles by using the same DCG P4 program that resulted in Figure 4.7 (100 entries and 256 bytes). However, to run the perf command is recommended to execute the program being analyzed with a single-core to exclude problems of complexities of managing multi-core issues, which we do not intend to analyze in this work. In Figure 4.7 we expose our results for both use cases (Outbound and Inbound) and packet I/O (DPDK and Socket-mmap) while OSNT is injecting packets for macsad to forward for a whole minute. In Figure 4.7 (a) we observe that as expected the "action_code_press" is the function with more cache-misses (21.54%) for the DPDK on the Inbound , it encapsulates the packet. Then, the "exact_lookup" consumes 13.41% by matching all tables parameters but lpm. On the other hand, Figure 4.7 (b) expose the same use case by using Socket-mmap, but with a different diagnosis, the function "action_code_nhop" is the one consuming most resources (17.69%), followed by "exact_lookup" (16.65%), "action_code_press" is the fourth one with much less percentage usage (9.42%). Surprisingly, on the Outbound we do not see the "pop" action, which removes the headers to be consuming many resources. In Figure 4.7 (c) we expose that the DPDK have many cache-misses while matching all the tables, this can be seen by functions: "lpm_lookup" (19.97%) and "exact_lookup" (12.99%). Lastly, in Figure 4.7 (d) we present the same results for socket-mmap, "exact_lookup" (22.80%) had most of the cache-misses, followed by the action "action_code_nhop" that sets the next hop (16.51%).



(a) perf cache-misses for DPDK on Inbound use case.

(b) perf cache-misses for Socket-mmap on Inbound use case.

(c) perf cache miss for DPDK on Outbound use case.

(d) perf cache miss for Socket-mmap on Outbound use case.

Figure 4.7: perf cache miss percentage per use cases and driver I/O.

4.3.2 Scalability

Considering that this DCG program would be responsible for being the gateway between a data-center and the internet, we expect to have hundreds, thousands or even millions of hosts connected at the same time. Thus, we understand that it is essential to evaluate the impact of increasing the entries on the switch.

In this scenario, traffic traces have different numbers (from 100 to 100K) of unique flows, randomly generated per use case experiment run but consistent across different packet sizes, limiting the impact of the lookup process and underlying caching system which would depend on the traffic pattern. We evaluate two different packets I/O drivers.

In Figure 4.8 and 4.9 we expose that there is a small overhead on performance when we increase the number of entries. As expected, we found an inversely proportional relationship between the number of entries and the throughput achieved; this is due to memory usage while matching the lpm.

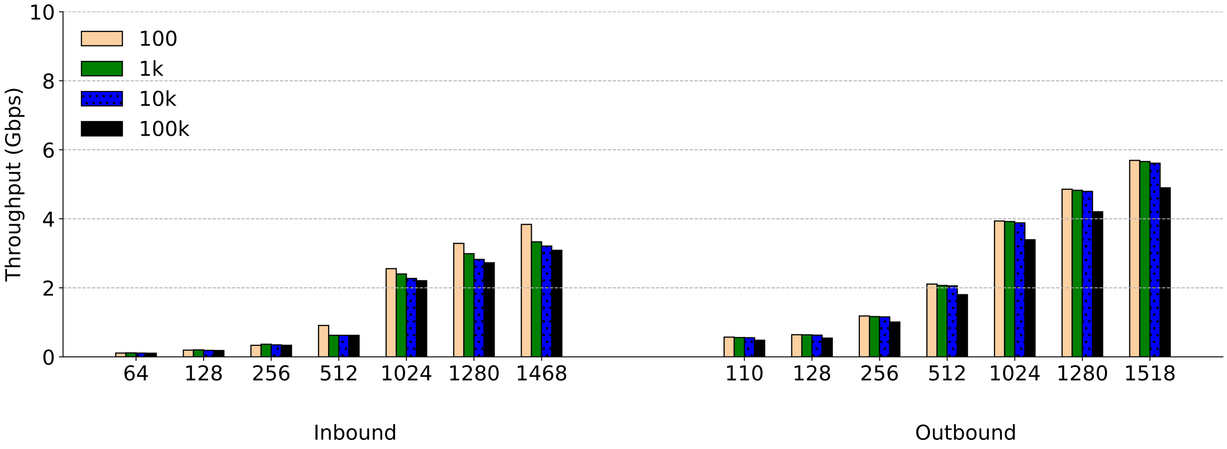


Figure 4.8: Impact of FIB sizes in the Throughput for DCG with Socket-mmap (four cores experiment).

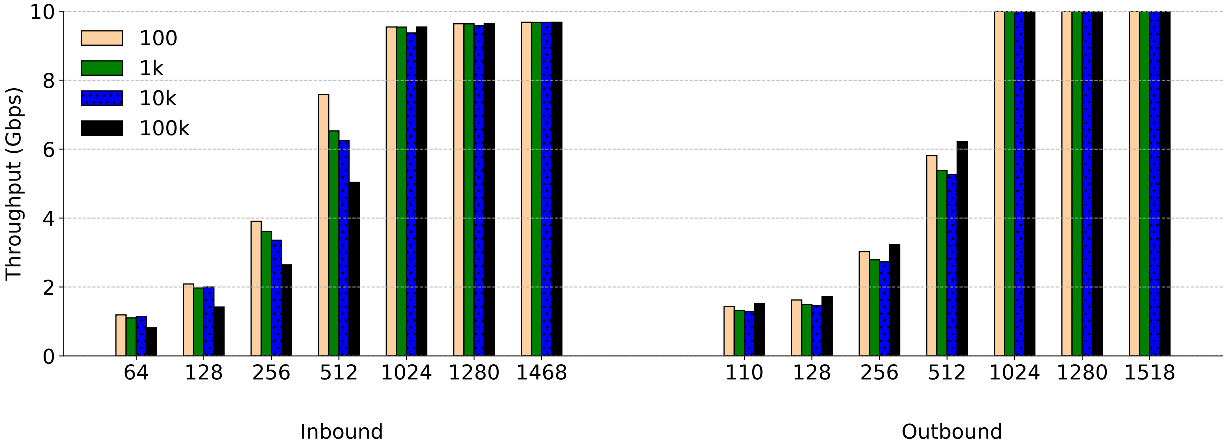
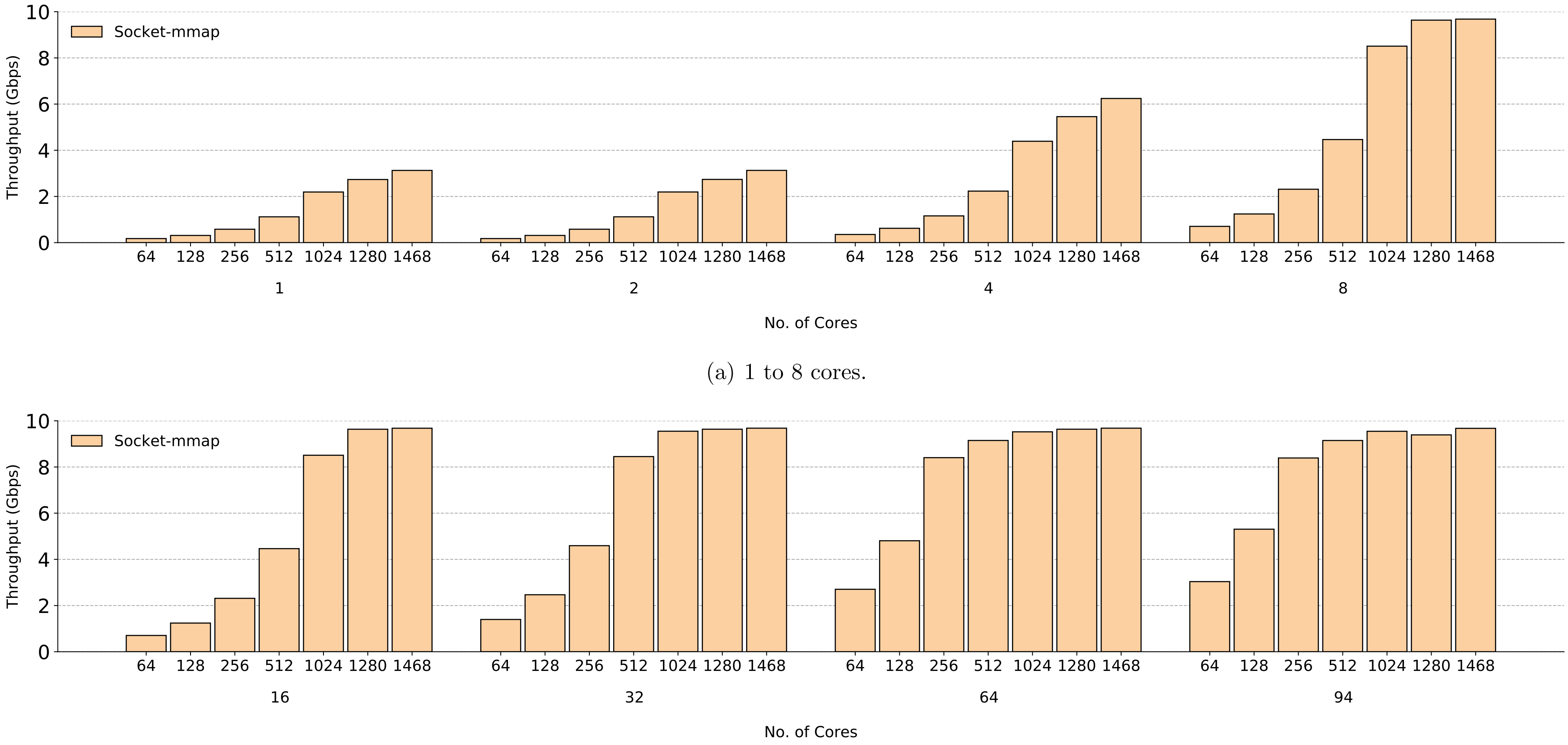


Figure 4.9: Impact of FIB sizes in the Throughput for DCG with DPDK (four cores experiment).

4.3.3 Multi-architecture

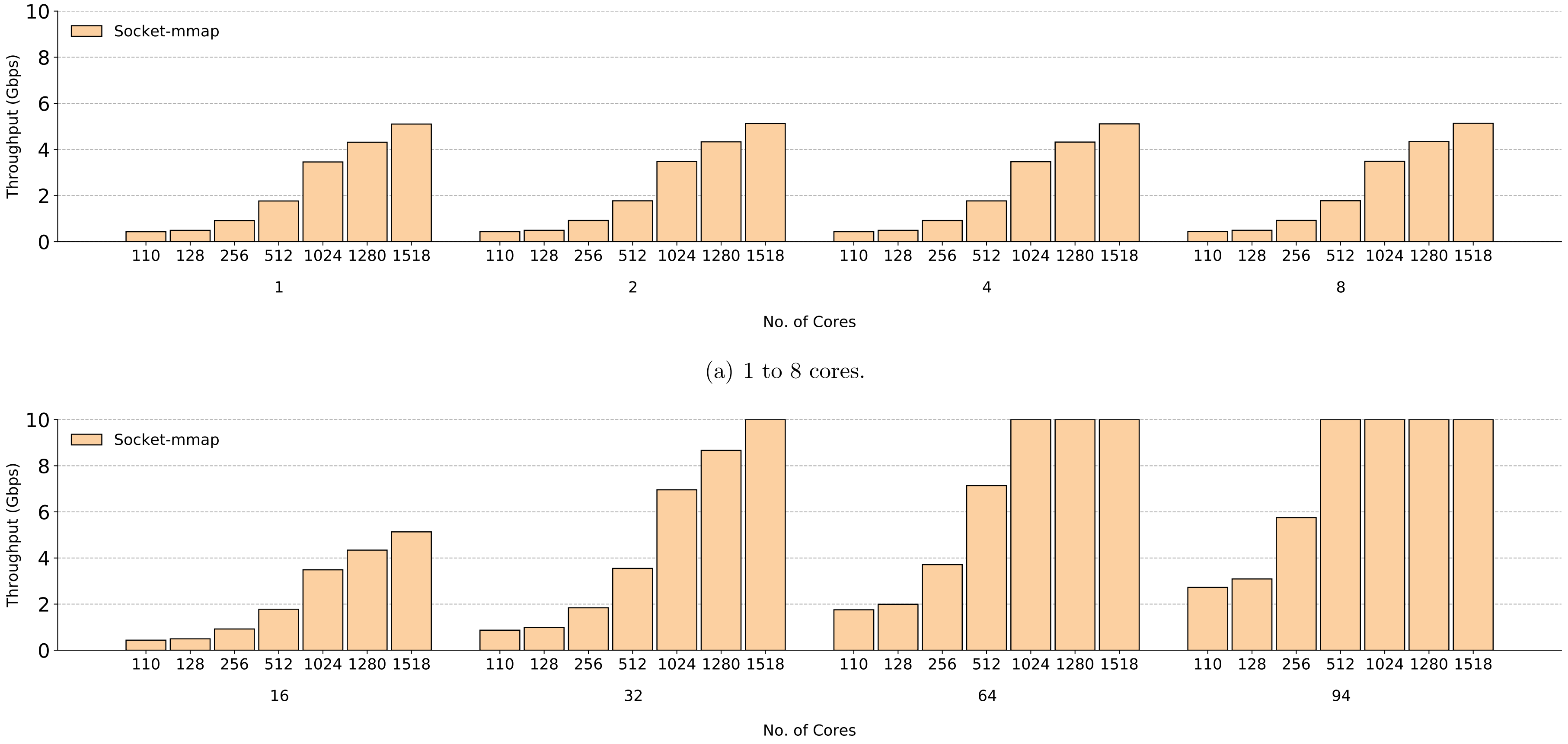
MACSAD has three main goals, **programmability**, **performance** and the support of **multiple architectures**. In this section, we explore the last one, until now we have exposed our VXLAN program running only on x86 servers with different configurations (varying memory, number of entries and cores). However, since MACSAD leverage on ODP APIs functions, it can be run on other architectures supported by ODP. In this example, our target is an ARM Cavium ThunderX System. In Figure 4.10 (a) and (b) we expose the performance for the Inbound use case, the X-axis represents the packet size and the number of cores used to achieve the throughput. Then, we evaluate the performance from 1 to 94 cores; the line rate was achieved for packets with more than 1280 bytes with just eight cores. However, by increasing the number of cores, we were able to achieve 3 Gbps for packets of 64 Bytes. In Figure 4.11 (a) and (b) we present the performance evaluation for the Outbound, an interesting fact is that different from the Inbound, from 1 to 16 cores the increases of cores did not alter the throughput. Unfortunately, we did not find a clear explanation of this behavior, and we consider further analysis necessary. Lastly, we found that the performance on both use cases with 94 cores achieved near the line rate for packets of 512 bytes.



(a) 1 to 8 cores.

(b) 16 to 94 cores.

Figure 4.10: Multi-core performance evaluation of Cavium Thunder X on Inbound use case



(a) 1 to 8 cores.

(b) 16 to 94 cores.

Figure 4.11: Multi-core performance evaluation of Cavium Thunder X on Outbound use case

4.4 Extended evaluation on load balancing performance

Considering our previous results, we found that both CRC32 and Checksum16 did not equally distribute the flows between our servers. Thus, we seek to extend our analysis for the behavior of some of the main polynomials that are not implemented on ODP APIs yet. Since MACSAD is restricted to ODP APIs, we were not able to execute this test using the compiler. Then, we made a few Python codes that performs the same Load Balance equation of the primitive implemented on MACSAD and a heatmap generator of the results. To choose the functions we to be analyzed, we base some recommended polynoms from an article from the Carnegie Mellon University of CRC selection for embedded networks (Koopman & Chakravarty 2004) and the most used polynomials (CRC8, CRC16, CRC32, and CRC32c). The BB-gen (Cesen & Patra 2018) code we modified to have a python script that generates 1000 *".txt"* files for IPv4 and IPv6 addresses, each one containing 1,048,576 random IPs, or 2^{20}. Then, in another script we calculate the polynomials using "crcmod" library, the polynomials that we have tested are:

- crc-8
- crc-16
- crc-32
- crc-32c
- 0x8d95
- 0x973afb51
- 0xd175

Considering the polynomial input, we apply our formula to measure the best Load Balance function. Our metric is composed of four equations: Equation 4.1 calculates the expected distribution, Equation 4.2 the difference of the expected and the real case distribution to the power of two, Equation 4.3 the *Root Mean Square Error (RMSE)* and Equation 4.4 normalize the RMSE found in Equation 4.3 by $log_2(IP)$. We apply the following equations in our search for the best load balancing algorithm. Lastly, we save the results of each file to generate statistics that include average, 95 percentile, maximum and minimum for each generated file considering the Normalized Root Mean Square Error (NRMSE) results. In Figure 4.12 we illustrate this process. In Figures 4.13 and 4.14 we expose our main results using 95 percentile methodology (discarding the 5% outliers best results), while on Appendix E we expose the results for Maximum, Minimum, Average and others 95 percentiles polynomials.

$$f(x)_{exp} = \frac{IP}{n} \tag{4.1}$$

$$\alpha = (f(x) - f(x)_{exp})^2 \tag{4.2}$$

$$RMSE = \sqrt{\frac{\sum_{x=0}^{x=n} \alpha}{n}} \tag{4.3}$$

$$NRMSE = \frac{RMSE}{\log_2(IP)} \tag{4.4}$$

Where:

- **IP:** Total of IPs sent;
- **f(x):** Is the distribution found by a specific conjunct of polynomial function, total hosts (Y-axis of Figures 4.13 and 4.14) and servers attending this hosts (X-axis of Figures 4.13 and 4.14)
- $f(x)_{exp}$: optimal IP distribution per server;
- **n:** The number of servers being balanced;
- **RMSE:** Root Mean Square Error;
- **NRMSE:** Normalized Root Mean Square Error;

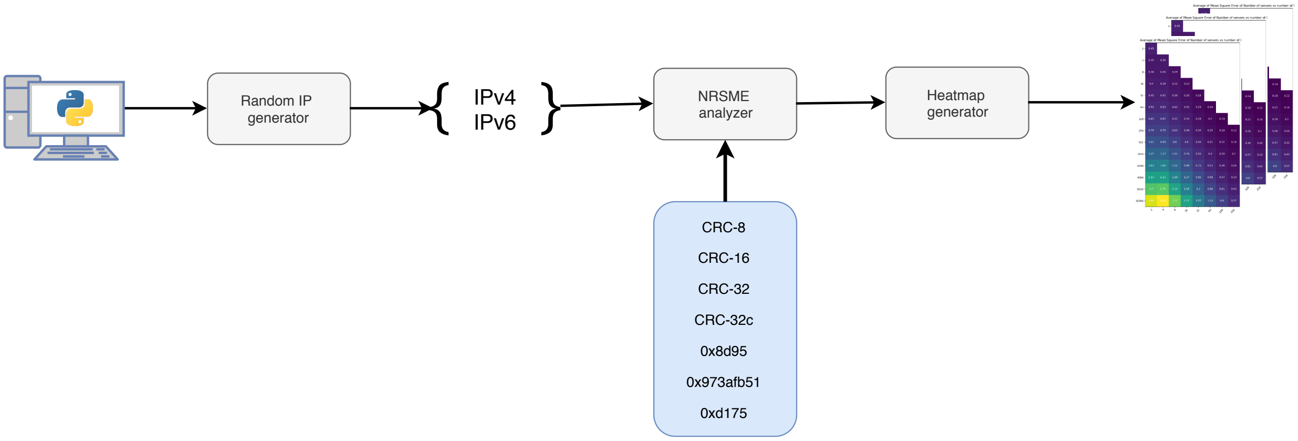
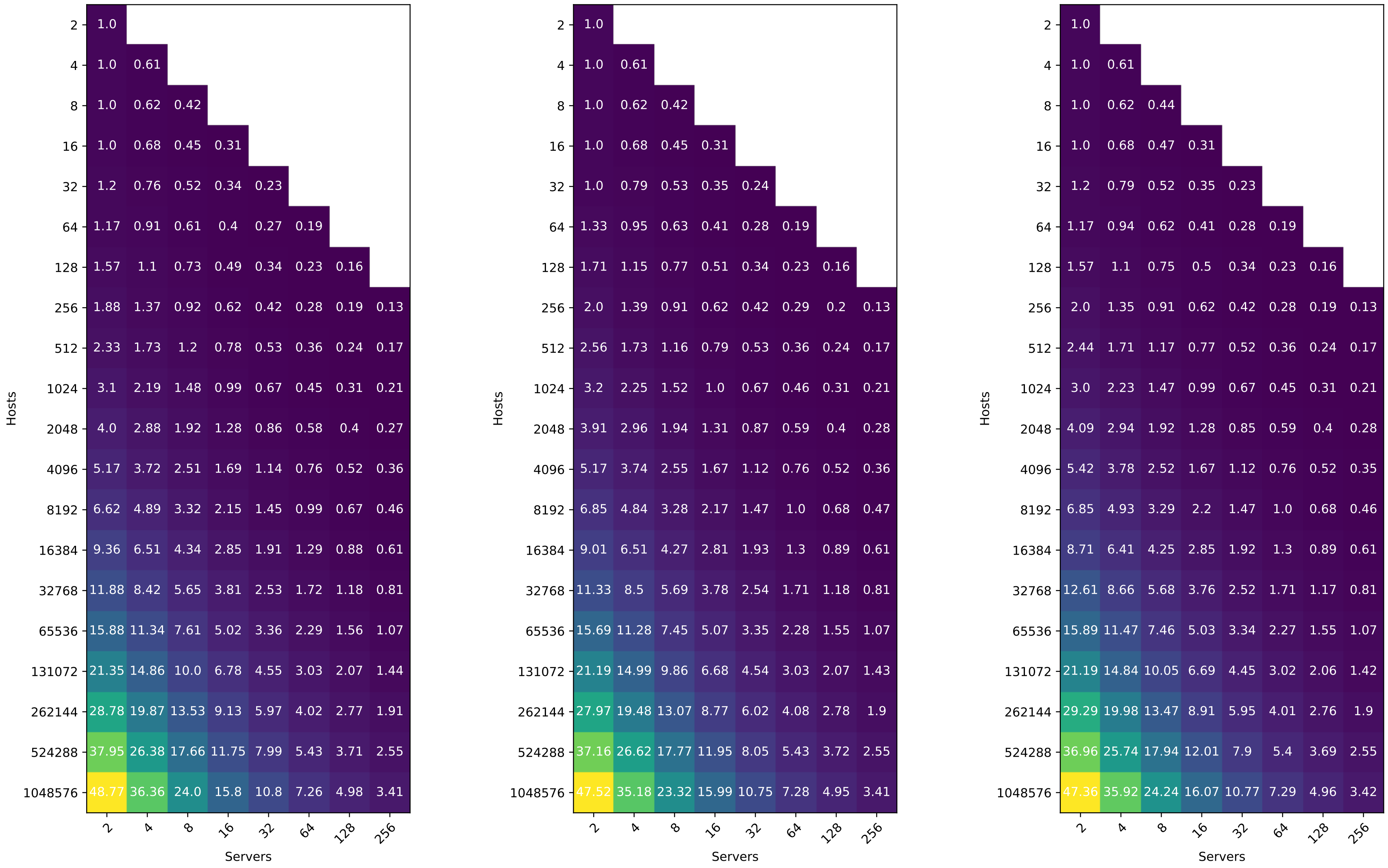


Figure 4.12: The Load Balancing extended evaluation script.

Considering that NRMSE measures how far the function distribution is from an equal distribution between servers, we compared the results by increasing the number of hosts tested and servers attending them. In Figures 4.13 we expose our results for IPv4 addresses, while in Figure 4.14 for IPv6. This experiment was conducted 1000 times for each polynomial with random IPs of up to 1,048,576. In the following Figures we present our analysis using the 95 percentile methodology. In general, we observe that an increase in the number of servers attending a fixed number of hosts, decreases the NRMSE, while an increase in the number of hosts with a fixed number of servers attending it increases NRMSE. Furthermore, we found that the 0xd175 give us the best result for the worst case, which is two servers with 1,048,576 hosts. Surprisingly, in most cases, CRC32c expose the best distribution, even better than CRC32, which in general is considered a more robust polynomial. Processors like AMD and Intel Atom do not have implemented CRC32 by default, which gives CRC32c performance advantage too, since it is cheaper in terms of computer cycles. In Figure 4.14 we observe similar results, but with a tiny lead to IPv4, this can be due to the fact that IPv6 is much larger than IPv4 and so it should require a higher CRC (e.g., CRC64).

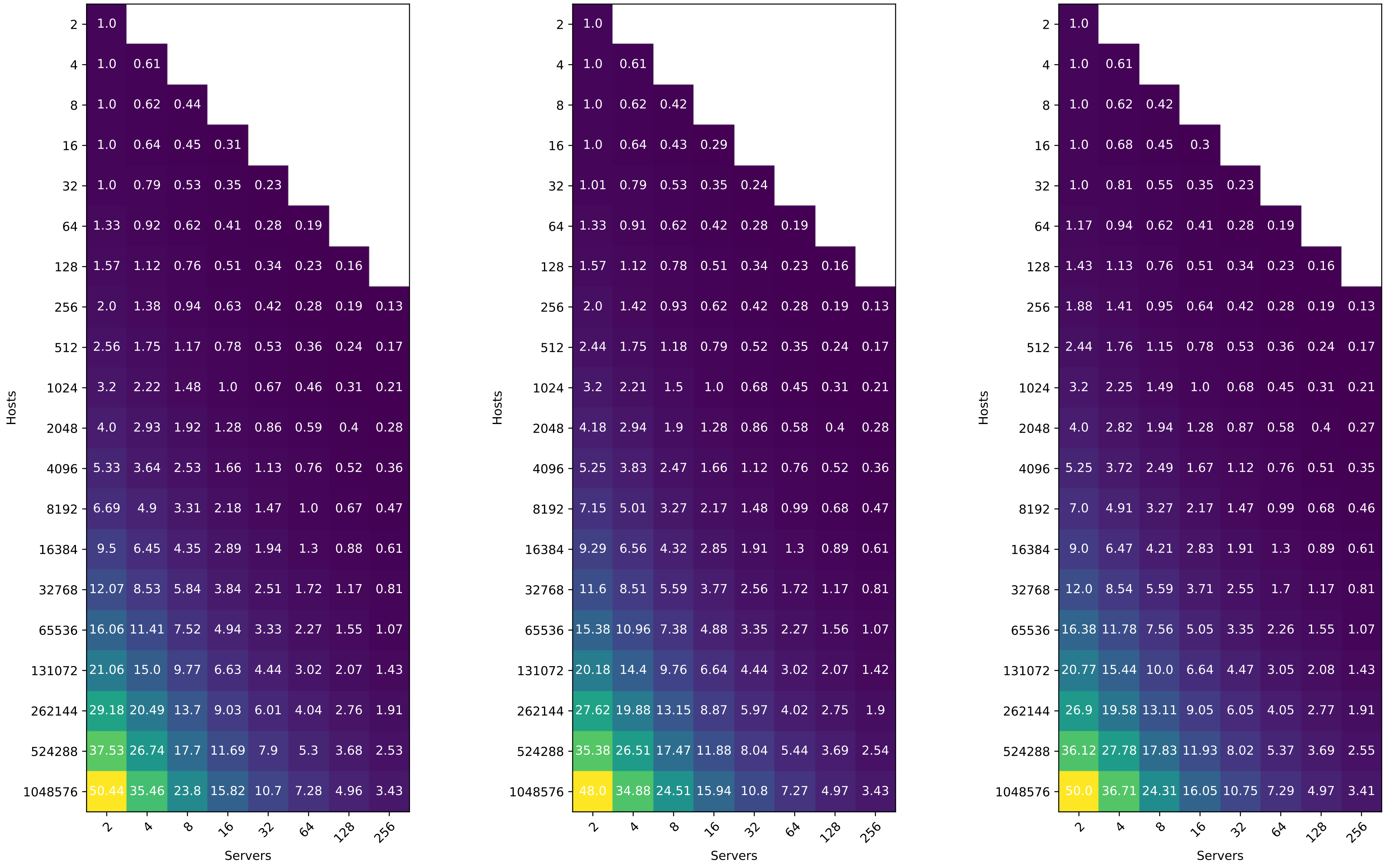


(a) crc-8

(b) crc-32c

(c) 0xd175

Figure 4.13: IPv4 95 percentile load balancing analysis



(a) crc-8 (b) crc-32c (c) 0xd175

Figure 4.14: IPv4 95 percentile load balancing analysis

Chapter 5

Conclusion and future work

This work has fulfilled its main objective, the design of a DCG VXLAN architecture implemented using P4 language that can be compiled to multi-architectures and still explore the best throughput capacity of each device. To address this solution we have: (i) implemented VXLAN DCG P4 program (ii) added support of new primitives to MACSAD, (iii) created an SDN controller to manage the packet traffic and table actions (iv) analyzed through a new metric the best polynomial function to perform our Load Balance feature, (v) carried performance and experimental evaluation of multi-core, scalability, and multi-architecture, and (vi) released all artifacts as open source.

This thesis describes the challenges we faced to achieve the DCG architecture. There, we evaluated the performance of use cases using two different packets I/O engines (DPDK and Socket_mmap). Through NFPA and OSNT we were able to transmit different packet sizes using PCAPs files. Comparing the different packet I/O drivers we can state that as expected the Linux default driver, or Socket-mmap, is much slower than DPDK.

These experiments expose two open source projects working together to achieve the same objective: to allow an open source dataplane programmability without compromising the network performance. ODP enables this goal by spreading a unique set of APIs for the dataplane, while P4 standardizes a common language to program the dataplane. The results obtained indicate what may be the next revolution after recent developments into Software Defined Networking (SDN) and Network Functions Virtualization (NFV), the data-plane programmability.

As future works, we consider a study of the same scenario but using Single Root I/O Virtualization (SR-IOV) in conjunction with MacS would be a nice fit since the DCG introduces an architecture based on VMs to analyze the latency and throughput. The SR-IOV bypass the hypervisor and allows its VMs to achieve near-line wire speed and low latency. Furthermore, in this work we analyzed the load balance distribution of different polynomials. We found that a more robust polynomial function result in general result in a good distribution. However, we do not consider the latency impact, it is expected that a more complex function will increase the latency. Considering the NFV technology, the DCG may be sliced by its network functions in a way that each P4 slice is managed by a centralized (and standardized) controller which could update its slices at runtime. Thus, by isolaing its network functions we would be able to run each one independently.

Through the results of this thesis, new questions appear, mainly on performance and portability. We noted that as expected there is a correlation between the number of matches/actions and the throughput of our program. Whippersnapper (Dang, Wang, Jepsen, Brebner, Kim, Rexford, Soulé & Weatherspoon 2017) start the discussion of performance impact and complexity of P4 programs. Since P4 programs allow full programmability of the dataplane, different P4 programs can achieve the same functionality, e.g., a table with two matches can be split into two tables without compromising the architecture, but it may increase the overhead. An analysis of the performance impact of the most critical functions is necessary to optimize the programs further.

We have tested the DCG on two servers, x86 and ARMv8. A future research may evaluate our test on other platforms, a NetFPGA or a Raspberry Pi. MACSAD still support a minimal number of primitives, allowing just a few use-cases to be tested. Thus, to have a full programmable dataplane network with support to multiple architectures, new primitives still need to be added. Another approach to be enhanced is the way control plane works on MACSAD, which by now use a non-standardized controller that needs to be manually written for each use-case. Furthermore, once a new functionality is added on the dataplane side, the same needs to be manually described on a simple controller, and then restart both of them. Aiming to solve this problem, P4 Runtime [1] surge as a silicon and protocol independent approach to auto-generate APIs and using Yet Another Next Generation (YANG) data-modeling language [2] to allow a smoother integration of P4 switches to controllers, some examples of applications can be seen on ONOS[3] and OpenDaylight[4] websites.

[1] https://github.com/p4lang/PI

[2] https://tools.ietf.org/html/rfc6020

[3] https://wiki.onosproject.org/display/ONOS/P4Runtime+support+in+ONOS

[4] https://wiki.opendaylight.org/view/P4$_{Plugin} : Main$

References

Antichi, G., Shahbaz, M., Geng, Y., Zilberman, N., Covington, A., Bruyere, M., McKeown, N., Feamster, N., Felderman, B., Blott, M., Moore, A. & Owezarski, P. (2014). OSNT: Open source network tester, *IEEE Network* (5): 6–12. http://yuba.stanford.edu/~nickm/papers/osnt.pdf.

Barham, P., Park, K., Weatherspoon, H., Zhou, L., Chase, J. & Dean, J. (2013). Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation NSDI'13, *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation NSDI'13* pp. 1–555. https://www.usenix.org/conference/nsdi13/tech-schedule/technical-sessions.

Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T. & Lantz, B. (2014). ONOS: towards an open, distributed SDN OS, *Proceedings of the third workshop on Hot topics in software defined networking - HotSDN '14* pp. 1–6. http://dl.acm.org/citation.cfm?id=2620728.2620744.

Bosshart, P., Daly, D., Izzard, M., McKeown, N., Rexford, J., Schlesinger, C., Talayco, D., Vahdat, A., Varghese, G. & Walker, D. (2013). Programming Protocol-Independent Packet Processors, **44**(3): 88–95. http://arxiv.org/abs/1312.1719.

Bradner, S. & McQuaid, J. (1999). Benchmarking methodology for network interconnect devices, *RFC 2544*, RFC Editor. http://www.rfc-editor.org/rfc/rfc2544.txt.

Cavium / XPliant® CNX880xx (2015). https://www.cavium.com/pdfFiles/CNX880XX_PB_Rev1.pdf?x=2.

Cesen, F. E. R. & Patra, P. G. K. (2018). BB-Gen : A Packet Crafter for Data Plane Evaluation. https://intrig.dca.fee.unicamp.br/wp-content/plugins/papercite/pdf/demo_bbgen_sigcomm_2018.pdf.

Csikor, L., Szalay, M., Sonkoly, B. & Toka, L. (2015). NFPA: Network function performance analyzer, *2015 IEEE Conference on Network Function Virtualization and Software Defined Network, NFV-SDN 2015* pp. 15–17. http://real.mtak.hu/40987/1/paper.pdf.

Dang, H. T., Wang, H., Jepsen, T., Brebner, G., Kim, C., Rexford, J., Soulé, R. & Weatherspoon, H. (2017). Whippersnapper: A P4 Language Benchmark Suite, *ACM Symposium on SDN Research (SOSR)* pp. 95–101. https://www.cs.princeton.edu/~jrex/papers/whippersnapper17.pdf.

Davie, B. & Gross, J. (2016). A Stateless Transport Tunneling Protocol for Network Virtualization (STT), IETF Draft. https://tools.ietf.org/html/draft-davie-stt-01.

de Melo, A. C. (2010). The New Linux 'perf' Tools, *Linux Kongress* . https://pdfs.semanticscholar.org/16ca/fd05fa375dfe370274cd22b4c16c72d6c53b.pdf.

Diedricks, I. (2015). Cisco extends market leadership for Unified Access with revolutionary ASIC. https://blogs.cisco.com/enterprise/cisco-extends-market-leadership-for-unified-access-with-revolutionary-asic.

Duncan, R. & Jungck, P. (2009). PacketC language for high performance packet processing, *2009 11th IEEE International Conference on High Performance Computing and Communications, HPCC 2009* pp. 450–457. https://ieeexplore.ieee.org/abstract/document/5167027.

Feamster, N., Rexford, J. & Zegura, E. (2014). The Road to SDN: An Intellectual History of Programmable Networks, *ACM Sigcomm Computer Communication* (2): 87–98. http://dl.acm.org/citation.cfm?id=2602204.2602219{&}coll=DL{&}dl=ACM{&}CFID=429855848{&}CFTOKEN=24281772.

Garg, P. & Wang, Y. (2015). Nvgre: Network virtualization using generic routing encapsulation, *RFC 7637*, RFC Editor. https://www.rfc-editor.org/rfc/pdfrfc/rfc7637.txt.pdf.

Intel® Ethernet Switch FM6000 Series (2017). https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-switch-fm6000-series-brief.pdf.

Kawashima, R., Nakayama, H. & Hayashi, T. (2017). Evaluation of Forwarding Efficiency in NFV-nodes toward Predictable Service Chain Performance, (4): 1–14. https://ieeexplore.ieee.org/document/7997907.

Koopman, P. & Chakravarty, T. (2004). Cyclic redundancy code (CRC) polynomial selection for embedded networks, pp. 145–154. http://users.ece.cmu.edu/~koopman/roses/dsn04/koopman04_crc_poly_embedded.pdf.

Kreutz, D., Ramos, F. M. V., Verissimo, P., Rothenberg, C. E., Azodolmolky, S. & Uhlig, S. (2014). Software-Defined Networking: A Comprehensive Survey, pp. 1–61. http://arxiv.org/abs/1406.0440.

Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M. & Wright, C. (2014). Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks, *RFC 7348*, RFC Editor. http://www.rfc-editor.org/rfc/rfc7348.txt.

McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S. & Turner, J. (2008). OpenFlow: Enabling Innovation in Campus Networks, *ACM SIGCOMM Computer Communication Review* (2): 69. http://portal.acm.org/citation.cfm?doid=1355734.1355746.

Medved, J., Varga, R., Tkacik, A. & Gray, K. (2014). OpenDaylight: Towards a model-driven SDN controller architecture, *Proceeding of IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks 2014, WoWMoM 2014* . https://ieeexplore.ieee.org/document/6918985/.

Open Networking Foundation (May 2015). Simplifying OpenFlow Interoperability with Table Type Patterns (TTP), ONF Solution Brief. https://3vf60mmveq1g8vzn48q2o71a-wpengine.netdna-ssl.com/wp-content/uploads/2014/10/sb-TTP.pdf.

Patra, P. G. & Rothenberg, C. (2016). MACSAD : Multi-Architecture Compiler System for Abstract Dataplanes (aka Partnering P4 with ODP), pp. 623–624. http://www.dca.fee.unicamp.br/~chesteve/pubs/2016-SIGCOMM-Demo-Mininet-MACSAD.pdf.

Patra, P. G., Rothenberg, C. E. & Pongracz, G. (2017). MACSAD: High performance dataplane applications on the move, *IEEE International Conference on High Performance Switching and Routing, HPSR* p. 6. http://www.dca.fee.unicamp.br/~chesteve/pubs/2017-06-IEEE-HPSR-MACSAD-Gyanesh.pdf.

Pepelnjak, I. (2012). Do we really need stateless transport tunneling (stt), http://blog.ipspace.net/2012/03/do-we-really-need-stateless-transport.html.

Pongracz, G., Molnar, L. & Kis, Z. L. (2013). Removing roadblocks from SDN: Openflow software switch performance on intel DPDK, *Proceedings - 2013 2nd European Workshop on Software Defined Networks, EWSDN 2013* pp. 62–67. https://ieeexplore.ieee.org/abstract/document/6680560.

Rahimi, R., Veeraraghavan, M., Nakajima, Y., Takahashi, H., Nakajima, Y., Okamoto, S. & Yamanaka, N. (2016). A high-performance OpenFlow software switch, *IEEE International Conference on High Performance Switching and Routing, HPSR* pp. 93–99. http://biblio.yamanaka.ics.keio.ac.jp/file/Reza_HPSR2016_1570252594.pdf.

Rizzo, L. (2012). NetMap: A Novel Framework for Fast Packet I/O, *2012 USENIX Annual Technical Conference* (257422): 101–112. https://www.usenix.org/system/files/conference/atc12/atc12-final186.pdf.

Robert Olsson (2005). Pktgen the Linux Packet Generator, *Proceedings of Linux Symposium* pp. 19–32. https://www.kernel.org/doc/ols/2005/ols2005v2-pages-19-32.pdf.

Schöne, R., Hackenberg, D. & Molka, D. (2012). Memory performance at reduced CPU clock speeds: an analysis of current x86_64 processors, *Proceedings of the USENIX Workshop on*

Power-Aware Computing and Systems (HotPower) . https://pdfs.semanticscholar.org/8668/5044b78aed871688f4c7e8d95b4b62538570.pdf.

Shahbaz, M., Choi, S., Pfaff, B., Kim, C., Feamster, N., McKeown, N. & Rexford, J. (2016). PISCES: A programmable, protocol-independent software switch, *2016 ACM Conference on Special Interest Group on Data Communication, SIGCOMM 2016* pp. 525–538. https://www.scopus.com/inward/record.uri?eid=2-s2.0-84986627816{&}partnerID=40{&}md5=a33dd327ed8989ce4e66ff76f2a6754d.

Singh, R. K., Chaudhari, N. S. & Saxena, K. (2012). Load Balancing in IP / MPLS Networks : A Survey, (May): 151–156. https://file.scirp.org/pdf/CN20120200011_57984796.pdf.

Sivaraman, A., Kim, C., Krishnamoorthy, R., Dixit, A. & Budiu, M. (2015). DC.p4, *Sosr* **4**: 1–8. http://dl.acm.org/citation.cfm?doid=2774993.2775007.

Song, H. (2013). Protocol-oblivious forwarding: unleash the power of SDN through a future-proof forwarding plane, *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking* pp. 127–132. http://dl.acm.org/citation.cfm?id=2491190.

Sridhar, T. & Wright, C. (2014). Geneve: Generic Network Virtualization Encapsulation draft-gross-geneve-00, pp. 1–46. https://tools.ietf.org/html/draft-gross-geneve-00.

Turull, D., Sjödin, P. & Olsson, R. (2016). Pktgen: Measuring performance on high speed networks, *Computer Communications* **82**: 39–48. http://kth.diva-portal.org/smash/get/diva2:919045/FULLTEXT01.pdf.

Voros, P. (2018). T4P4S : A Target-independent Compiler for Protocol-independent Packet Processors, *Ieee Hpsr* (June). https://www.researchgate.net/publication/326652427_T4P4S_A_Target-independent_Compiler_for_Protocol-independent_Packet_Processors.

Appendix A

Publications

- 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, June 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. In: IEEE COMSOC JSAC'18 Special Issue on Scalability Issues and Solutions for Software Defined Networks, December 2018

- **Feferman, D.**, Rothenberg, C. E. (2017). Modeling P4 programmable devices using YANG,4. In: X Encontro de Alunos e Docentes do DCA/FEEC/UNICAMP (EADCA). October 2017.

- Sebastian, J., Vallejo, M., **Feferman, D. L.**, & Rothenberg, C. E. (2018). Network Address Translation using a Programmable Dataplane Processor. In: 17º Workshop em Desempenho de Sistemas Computacionais e de Comunicação, September 2017.

- **Feferman, D.**, Unicamp, F., Sebastian, J., Unicamp, M., Franklin, N., Sousa, S. De, & Esteve, C. (2018). Uma Nova Revolução em Redes: Programação do Plano de Dados com P4. In: ERIPI, May 2018.

Appendix B

The DCG P4 code

```
//----------header----------//

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

header ethernet_t ethernet;

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

header ipv4_t ipv4;

header_type udp_t {
  fields {
    srcPort : 16;
    dstPort : 16;
    length_ : 16;
    checksum : 16;
  }
}

```

```
header udp_t udp;

header_type vxlan_t {
  fields {
    flags : 8;
    reserved : 24;
    vni : 24;
    reserved2 : 8;
  }
}

header vxlan_t vxlan;

header_type arp_t {
  fields {
  htype : 16;
  ptype : 16;
  hlength : 8;
  plength: 8;
  opcode: 16;
  }
}

header arp_t arp;
header ethernet_t inner_ethernet;
header ipv4_t inner_ipv4;

//--------parser----------//

#define MAC_LEARN_RECEIVER      1024
#define ETHERTYPE_IPV4          0x0800
#define ETHERTYPE_ARP           0x0806

#define IP_PROTOCOLS_IPHL_UDP   0x511
#define IP_UDP                  0x11
#define UDP_PORT_VXLAN          4789

#define BONE                    1
#define BTWO                    2
#define BTHREE                  3

#define BIT_WIDTH               16

parser start {
  return parse_ethernet;
}

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

```
 parser parse_arp{
  extract(arp);
  return ingress;
}

 parser parse_ipv4 {
  extract(ipv4);
    return select(ipv4.protocol){
    IP_UDP : parse_udp;
    default: ingress;
  }
}

 parser parse_udp {
  extract(udp);
  return select (latest.dstPort) {
    UDP_PORT_VXLAN : parse_vxlan;
    default : ingress;
  }
}

 parser parse_vxlan {
  extract(vxlan);
  return parse_inner_ethernet;
}

 parser parse_inner_ethernet {
  extract(inner_ethernet);
  return select(latest.etherType) {
    ETHERTYPE_IPV4 : parse_inner_ipv4;
    default: ingress;
  }
}

 parser parse_inner_ipv4 {
  extract(inner_ipv4);
  return ingress;
}

//--------action--------//

 action _drop() {
  drop();
}

 action _nop() {
}

field_list mac_learn_digest {
  ethernet.srcAddr;
  routing_metadata.ingress_port;
}

field_list inner_ipv4_checksum_list {
  inner_ipv4.version;
  inner_ipv4.ihl;
  inner_ipv4.diffserv;
  inner_ipv4.totalLen;
  inner_ipv4.identification;
  inner_ipv4.flags;
  inner_ipv4.fragOffset;
  inner_ipv4.ttl;
  inner_ipv4.protocol;
  inner_ipv4.srcAddr;
  inner_ipv4.dstAddr;
}

action mac_learn() {
  generate_digest(MAC_LEARN_RECEIVER, mac_learn_digest);
}

table MAClearn {
  reads {
    ethernet.srcAddr : exact;
  }
  actions {
    mac_learn;
    _nop;
  }
  size : 100;
}

header_type routing_metadata_t {
  fields {
    res: 2;
    aux : 2;
    ingress_port : 8;
    lb_hash: 16;
    mcast_grp : 4;
  }
}

metadata routing_metadata_t routing_metadata;

action forward(port, mac) {
  modify_field(standard_metadata.egress_port, port);
  modify_field(ethernet.dstAddr, mac);
  modify_field(routing_metadata.res, BTHREE);
}

action Tcast() {
  modify_field(routing_metadata.mcast_grp, 1);
  modify_field(routing_metadata.res, BONE);
}

action Tmac() {
  modify_field(routing_metadata.res, BTWO);
}

table MACfwd {
  reads {
    ethernet.dstAddr : exact;
  }
  actions {
    forward;
    _nop;
    _drop;
    Tcast;
    Tmac;
  }
  size : 100;
}

action arp() {
  generate_digest(ETHERTYPE_ARP, mac_learn_digest);
  modify_field(routing_metadata.res, BONE);
}

table ARPselect {
  reads {
    ethernet.etherType: exact;
  }
  actions {
    arp;
    _nop;
  }
  size : 2;
}

field_list load_balancer_fields {
  ipv4.srcAddr;
}

field_list_calculation load_hash {
  input {
    load_balancer_fields;
  }
  algorithm : csum16;
  output_width : BIT_WIDTH;
}

action balancer(){
  modify_field(routing_metadata.aux, BONE);
  modify_field_with_hash_based_offset(routing_metadata.lb_hash, 0, load_hash, 2);
}

action _pop(){
  modify_field(routing_metadata.aux, BTWO);
}

action jump(){
  modify_field(routing_metadata.aux, BTHREE);
}

table LBselector{
  reads {
    ipv4.dstAddr : exact;
  }
  actions {
    jump;
    _pop;
    balancer;
    _nop;
  }
  size: 100;
}

action _pop_vxlan(mac_dst, mac_src){
  modify_field(inner_ethernet.dstAddr, mac_dst);
  modify_field(inner_ethernet.srcAddr, mac_src);
  remove_header(ethernet);
  remove_header(ipv4);
  remove_header(vxlan);
  remove_header(udp);
}

table vpop{
  reads {
    inner_ipv4.dstAddr : exact;
  }
  actions {
    _pop_vxlan;
    _nop;
  }
  size: 100;
}

action press(vnid, srcAddr){

  add_header(vxlan);
  add_header(udp);
  add_header(inner_ipv4);
  copy_header(inner_ipv4, ipv4);
  add_header(inner_ethernet);
  copy_header(inner_ethernet, ethernet);
  modify_field(inner_ipv4.srcAddr, srcAddr);
  modify_field(inner_ipv4.protocol, 0x11);
  modify_field(inner_ipv4.ttl, 64);
  modify_field(inner_ipv4.version, 0x4);
  modify_field(inner_ipv4.ihl, 0x5);
  modify_field(inner_ipv4.identification, 1);
  modify_field(inner_ethernet.etherType, ETHERTYPE_IPV4);
  modify_field(udp.dstPort, UDP_PORT_VXLAN);
  modify_field(udp.srcPort, UDP_PORT_VXLAN);
  modify_field(udp.checksum, 0);
  modify_field(udp.length_, 140);
  modify_field(inner_ipv4.totalLen, 160);
  modify_field(vxlan.flags, 0x8);
  modify_field(vxlan.reserved, 0);
  modify_field(vxlan.vni, vnid);
  modify_field(vxlan.reserved2, 0);

}

```

```
table LB{
  reads {
    ipv4.srcAddr : exact;
  }
  actions {
    press;
    _nop;
  }
  size:100;
}

action nhop_ipv4(nhop_ipv4, dmac, macS) {
  modify_field(inner_ipv4.dstAddr, nhop_ipv4);
  modify_field(ethernet.dstAddr, dmac);
  modify_field(inner_ethernet.dstAddr, macS);
}

table LBipv4 {
  reads {
    routing_metadata.lb_hash : exact;
  }
  actions {
    nhop_ipv4;
    _nop;
  }
  size:100;
}

action nhop(port){
  modify_field(standard_metadata.egress_port, port);
  modify_field(inner_ipv4.ttl,ipv4.ttl - 1);
}

table vxlan{
  reads {
    vxlan.vni : exact;
  }
  actions {
    _nop;
  }
}

table L3{
  reads {
    inner_ipv4.dstAddr : lpm;
  }
  actions {
    nhop;
    _nop;
  }
  size:100;
}

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

```

table sendout {
  reads {
    standard_metadata.egress_port : exact;
  }
  actions {
    _nop;
    rewrite_src_mac;
  }
  size : 100;
}

//--------control--------//

control ingress {
  apply(MAClearn);
  apply(MACfwd);
  if (routing_metadata.res == BTWO){
    apply(ARPselect);
    if (routing_metadata.res == BTWO){
        apply(LBselector);

        if (routing_metadata.aux == BONE){
          apply(LB);
          apply(LBipv4);
        }
        apply(vxlan);
        apply(L3);
        apply(sendout);
        if (routing_metadata.aux == BTWO){
          apply(vpop);
        }
    }
  }
}

control egress {
}
```

Listing B.1: The DCG P4 code.

Appendix C

P4 graphs

This appendix present graphs of the parser representation and the tables dependencies of the VXLAN P4 program.

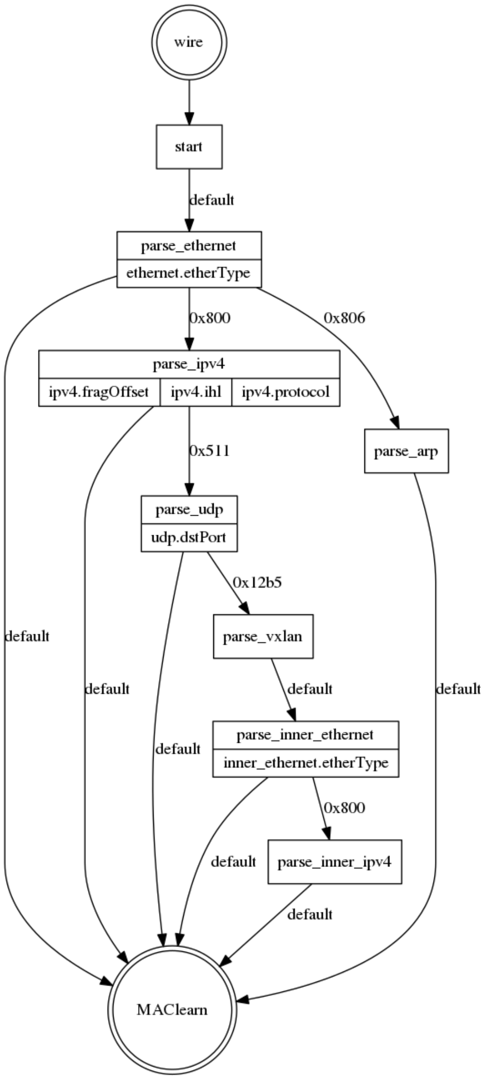


Figure C.1: The P4 parser representation of the VXLAN program.

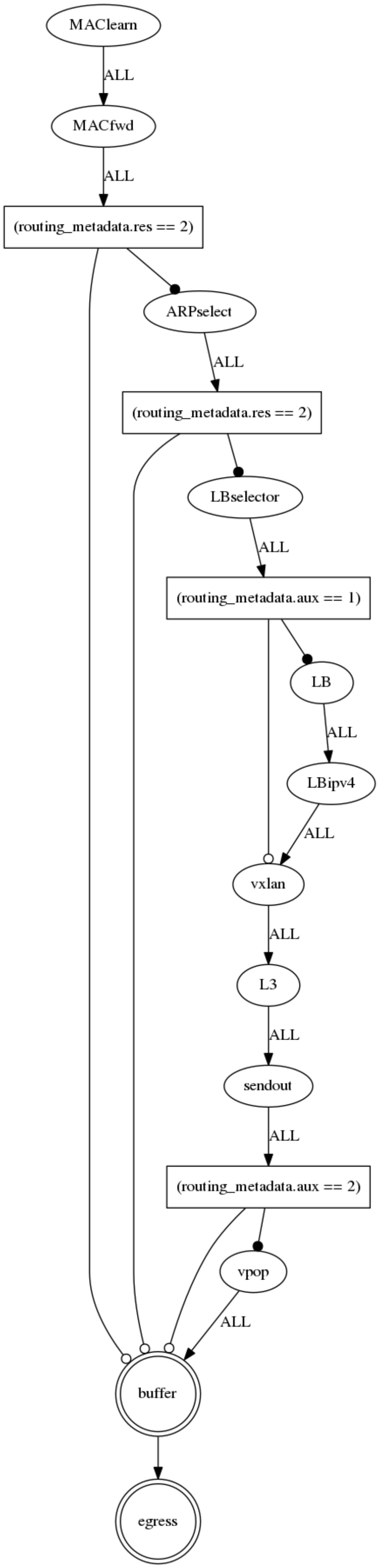

MAClearn
ALL
MACfwd
ALL
(routing_metadata.res == 2)
ARPselect
ALL
(routing_metadata.res == 2)
LBselector
ALL
(routing_metadata.aux == 1)
LB
ALL
LBipv4
ALL
vxlan
ALL
L3
ALL
sendout
ALL
(routing_metadata.aux == 2)
vpop
ALL
buffer
egress

Figure C.2: The P4 tables dependencies representation of the VXLAN program.

Appendix D

The Load Balancing test code

Bellow we present our code to analyze the distribution of Load Balancing functions. The test is composed of three steps:

1. Modify the code from BB-gen to generate random IPs.

2. Test each group of IP through our RMSE error methodology seeking the algorithm with the best distribution. Since we are working with a large number of IPs we included the multithreading feature to speed up the process.

3. Compile the results into heatmaps graphs.

```
import os
import random
from random import shuffle
import argparse
import math
import random
import threading

parser = argparse.ArgumentParser(description='IPv4 PCAP generator.')
args = parser.parse_args()
rep = args.num
def generate(start, rep):
    r = []
    i = 0
    for z in range(start, rep):
            for i in range(1,254):
                r.append(i)
            shuffle(r)
            for m in range(1048576):
                l = 0
                ip_c = ""
                for i in range(4):
                    if l == 1:
                        ip_c = ip_c + "." + str(r[0])
                        l = 0
                    else:
                        ip_c = ip_c + str(r[0])
                l = l + 1

                shuffle(r)
            os.system("echo " + str(ip_c) + " >> ./ipv4/ip_" + str(z) + ".txt")
        r = []
    print z



thread1 = threading.Thread(target=generate, args=(0, rep,))
try:
    thread1.start()
```

Listing D.1: The IP generation code.

```
import CRCmod.predefined
import csv
import zlib
import matplotlib
import hashlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import time
import math
import os
import sys
import threading
first=sys.argv[1]
second=sys.argv[2]
hosts = [2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288]

def get_hash(j, first, second):
    s = (int(second), int(len(hosts)))
    matrix = np.zeros(s)
    for k in range (int(first),int(second)):
        with open('./ips/ip_' + str(k) + '.txt', 'r') as fd:
            x = 0
            cont = 0
            CRC32 = []
            for row in fd:
                cont = cont + 1
                if cont > 524288:
                    break
                p = zlib.CRC32(row) & 0xffffffff
                q = int(p) % j
                CRC32.append(q)
                if cont == int(hosts[x]):
                    if cont < j:
                        continue
                    i = hosts[x]
                    result = np.bincount(CRC32)
                    avg = i/j
                    hit = abs(result - avg)
                    eqm = 0
```

```
                                        for m in hit:
                                                eqm = m**2+eqm
                                        eqm = eqm/float(j)
                                        lista[hosts.index(i),servers.index(j)] = round(((math.sqrt(eqm))/math.log(i,2)),2)
                                        if x < int(len(hosts))-1:
                                                x = x + 1
                                        else:
                                                break

thread1 = threading.Thread(target=get_hash, args=(2, first, second,))
thread2 = threading.Thread(target=get_hash, args=(4, first, second,))
thread3 = threading.Thread(target=get_hash, args=(8, first, second,))
thread4 = threading.Thread(target=get_hash, args=(16, first, second,))
thread5 = threading.Thread(target=get_hash, args=(32, first, second,))
thread6 = threading.Thread(target=get_hash, args=(64, first, second,))
thread7 = threading.Thread(target=get_hash, args=(128, first, second,))
thread8 = threading.Thread(target=get_hash, args=(256, first, second,))

thread1.start()
time.sleep(120)
thread2.start()
time.sleep(120)
thread3.start()
time.sleep(120)
thread4.start()
time.sleep(120)
thread5.start()
time.sleep(120)
thread6.start()
time.sleep(120)
thread7.start()
time.sleep(120)
thread8.start()
```

Listing D.2: The RMSE error code.

```
import csv
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import math
import time
lista = []
hosts = ["2", "4", "8", "16", "32", "64", "128", "256", "512", "1024", "2048", "4096", "8192", "16384", "32768", "65536", "131072", "262144", "524288"]
servers = ["2", "4", "8",
           "16", "32", "64", "128", "256"]
perc = np.zeros((len(hosts), len(servers)))
maximo = np.zeros((len(hosts), len(servers)))
minimo = np.zeros((len(hosts), len(servers)))
avg = np.zeros((len(hosts), len(servers)))
for i in range(len(hosts)):
        for j in range(len(servers)):
                print "servers = " + str(servers[j])
                print "hosts = " + str(hosts[i])
        if int(servers[j]) > int(hosts[i]):
            perc[i,j] = np.nan
            maximo[i,j] = np.nan
            minimo[i,j] = np.nan
            avg[i,j] = np.nan
            continue
        csvFile = csv.reader(open(”./lista_ipv4/lista[” + str(i) + ”,” + str(j) + ”].txt”, ”rb”))
        print ”./lista/lista[” + str(i) + ”,” + str(j) + ”].txt”
        for row in csvFile:
            lista.append(float(row[0]))
        total = 0
        percentile = np.array(lista)
        perc[i,j] = round(np.percentile(percentile, 95),2)
        maximo[i,j] = max(lista)
        minimo[i,j] = min(lista)
        for k in range(len(lista)):
            total = total + lista[k]
        avg[i,j] = round(total/len(lista),2)
        lista = []
harvest = avg


harvest.astype(int)
print harvest
fig, ax = plt.subplots(figsize=(10,10))
im = ax.imshow(harvest)

ax.set_xticks(np.arange(len(servers)))
ax.set_yticks(np.arange(len(hosts)))

ax.set_xticklabels(servers)
ax.set_yticklabels(hosts)


plt.setp(ax.get_xticklabels(), rotation=45, ha=”right”,
         rotation_mode=”anchor”)


for i in range(len(hosts)):
    for j in range(len(servers)):
        text = ax.text(j, i, harvest[i, j],
                       ha=”center”, va=”center”, color=”w”)

ax.set_title(”Average of Mean Square Error of Number of servers vs number of IPs”)
fig.tight_layout()
plt.show()
plt.savefig(”./gyn/ipv4/avg.png”)

fig.clf()
harvest = maximo
harvest.astype(int)
print harvest
fig, ax = plt.subplots(figsize=(10,10))
im = ax.imshow(harvest)
```

```

ax.set_xticks(np.arange(len(servers)))
ax.set_yticks(np.arange(len(hosts)))

ax.set_xticklabels(servers)
ax.set_yticklabels(hosts)

plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
         rotation_mode="anchor")

for i in range(len(hosts)):
    for j in range(len(servers)):
        text = ax.text(j, i, harvest[i, j],
                       ha="center", va="center", color="w")

ax.set_title("Max of Mean Square Error of Number of servers vs number of IPs")
fig.tight_layout()
plt.show()
plt.savefig("./gyn/ipv4/max.png")

fig.clf()
harvest = minimo
harvest.astype(int)
print harvest
fig, ax = plt.subplots(figsize=(10,10))
im = ax.imshow(harvest)

ax.set_xticks(np.arange(len(servers)))
ax.set_yticks(np.arange(len(hosts)))

ax.set_xticklabels(servers)
ax.set_yticklabels(hosts)

plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
         rotation_mode="anchor")

for i in range(len(hosts)):
    for j in range(len(servers)):
        text = ax.text(j, i, harvest[i, j],
                       ha="center", va="center", color="w")

ax.set_title("Min of Mean Square Error of Number of servers vs number of IPs")
fig.tight_layout()
plt.show()
plt.savefig("./gyn/ipv4/min.png")

fig.clf()
harvest = perc
harvest.astype(int)
print harvest
fig, ax = plt.subplots(figsize=(10,10))
im = ax.imshow(harvest)
ax.set_xticks(np.arange(len(servers)))
ax.set_yticks(np.arange(len(hosts)))
ax.set_xticklabels(servers)
ax.set_yticklabels(hosts)
```

```

plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
         rotation_mode="anchor")

 for i in range(len(hosts)):
     for j in range(len(servers)):
         text = ax.text(j, i, harvest[i, j],
                        ha="center", va="center", color="w")

ax.set_title("95 percentile of Mean Square Error of Number of servers vs number of IPs")
fig.tight_layout()
plt.show()
plt.savefig("./gyn/ipv4/perc.png")
```

Listing D.3: The Load Balancing heatmaps.

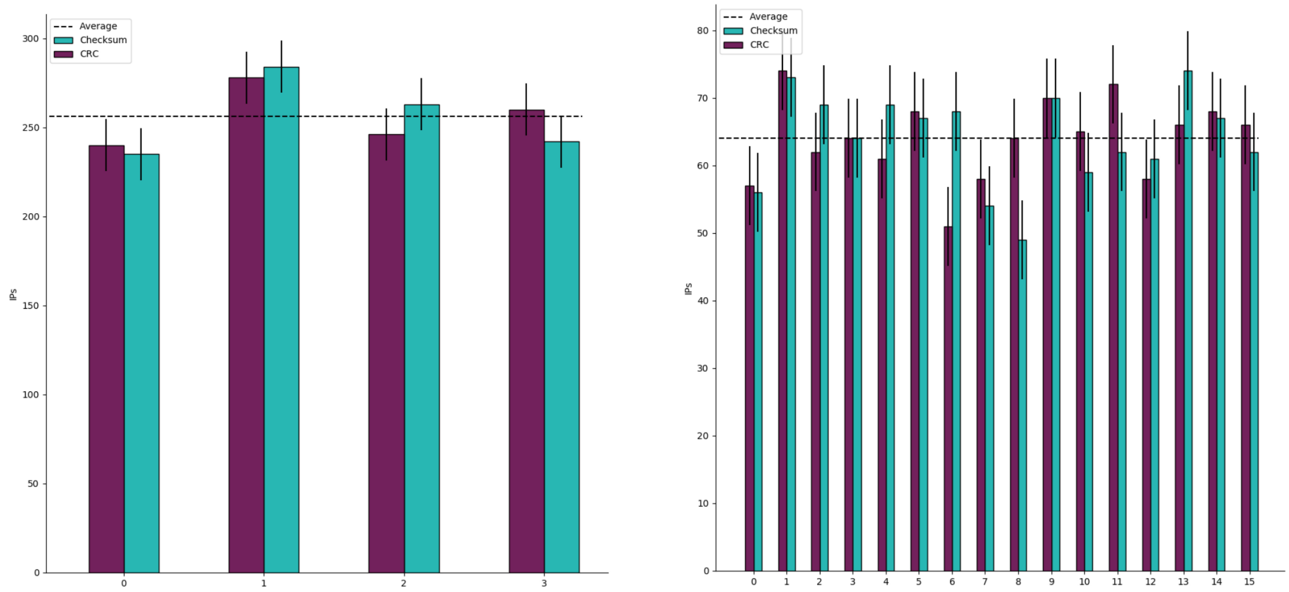
Appendix E

The LB analysis

In this Appendix we present our analysis of the Load Balancing feature through multiples polynomials.

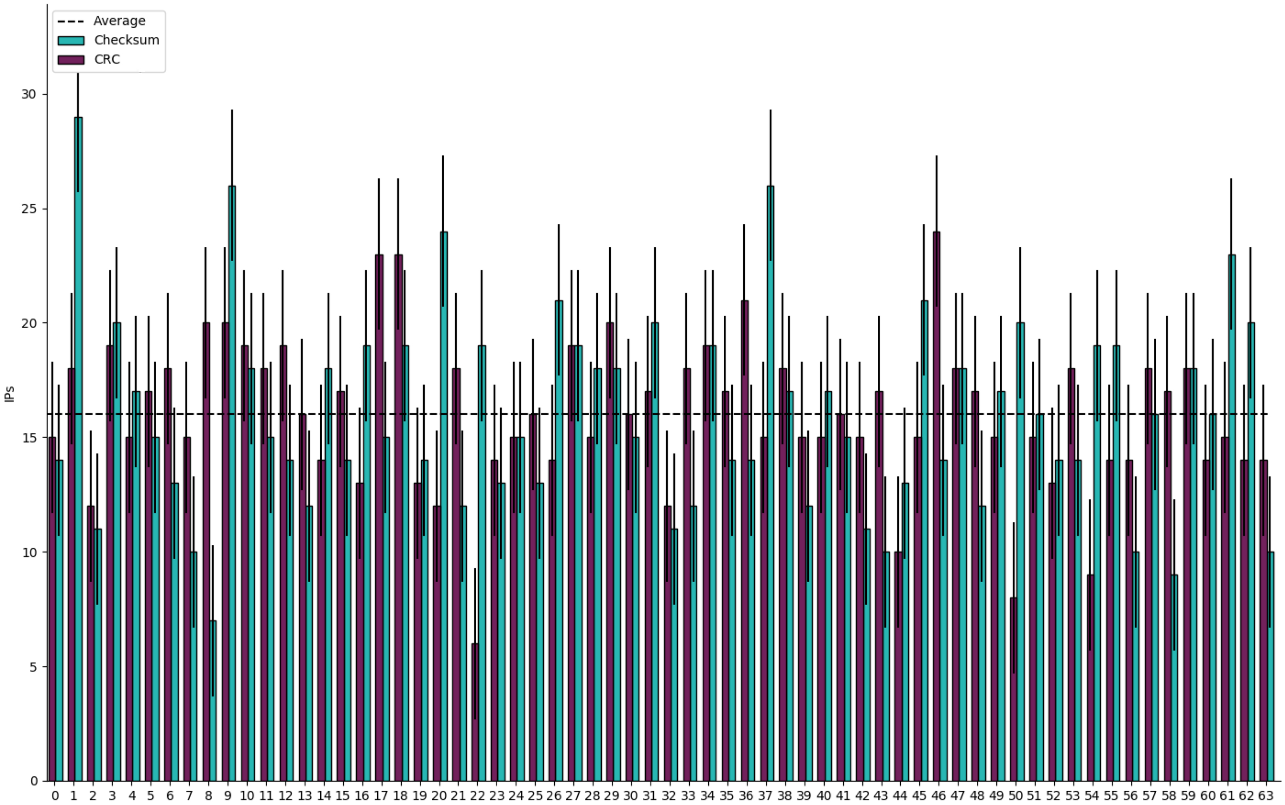
E.1 Functional evaluation

In this section we present the comparison of CRC32 with Checksum considering two others PCAPs of 1024 entries.



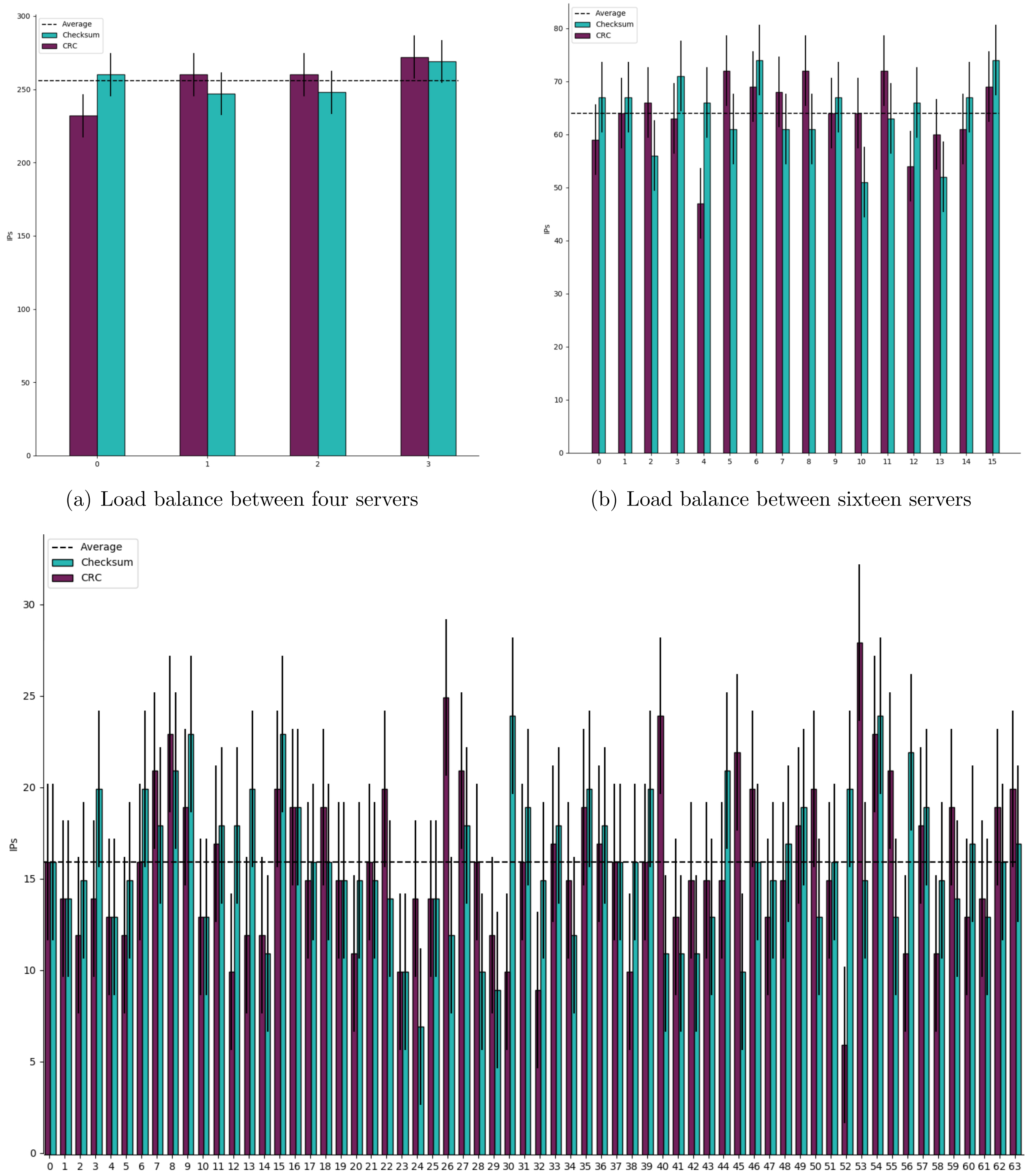
(a) Load balance between four servers

(b) Load balance between sixteen servers



(c) Load balance between sixty-four servers

Figure E.1: Scenario 2 PCAP files Load Balanced.



(a) Load balance between four servers

(b) Load balance between sixteen servers

(c) Load balance between sixty-four servers

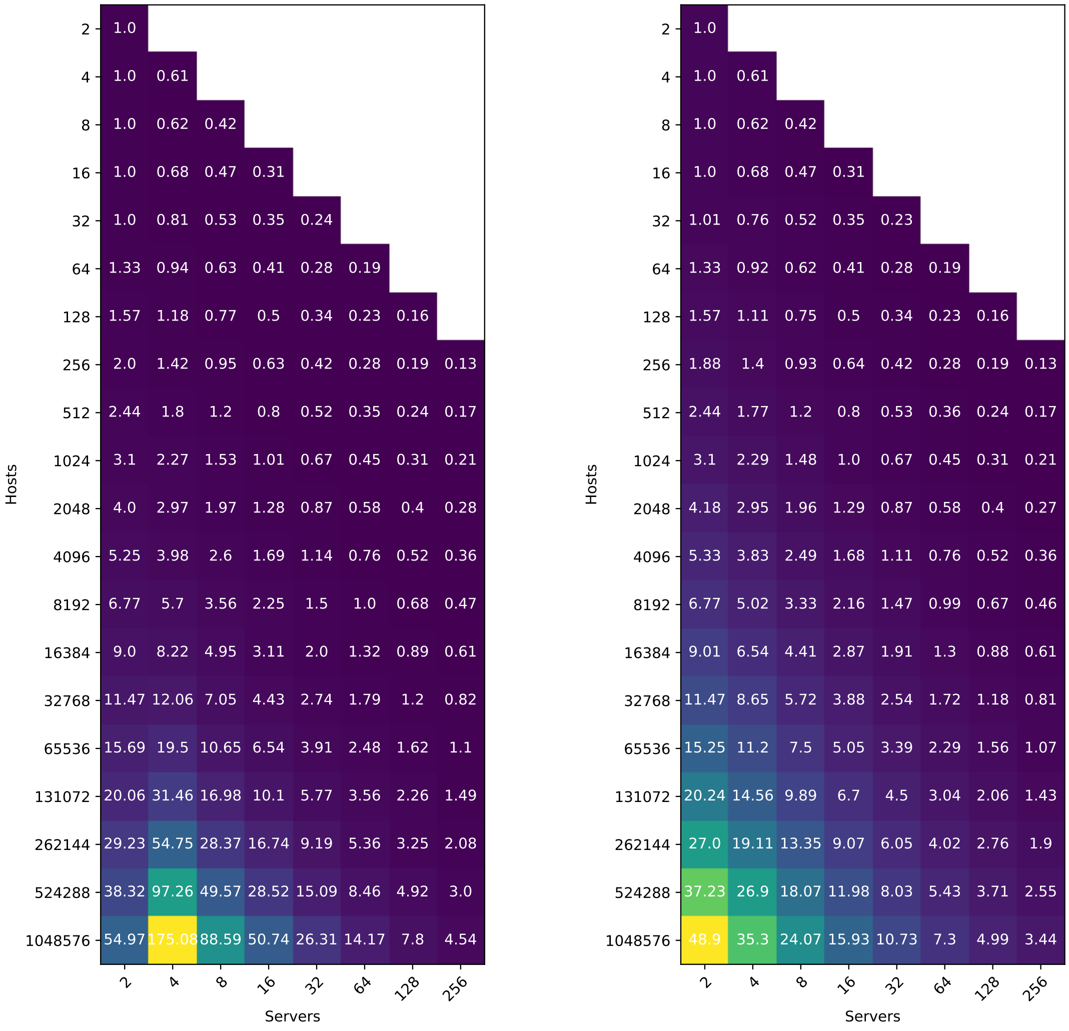
Figure E.2: Scenario 3 PCAP files Load Balanced.

E.2 Automated LB analysis

In this chapter we present others measures on our Load Balancing functions analysis. We have considered three additional parameters for the error measure, they are: the average, the maximum the minimum of each square in the heatmap.

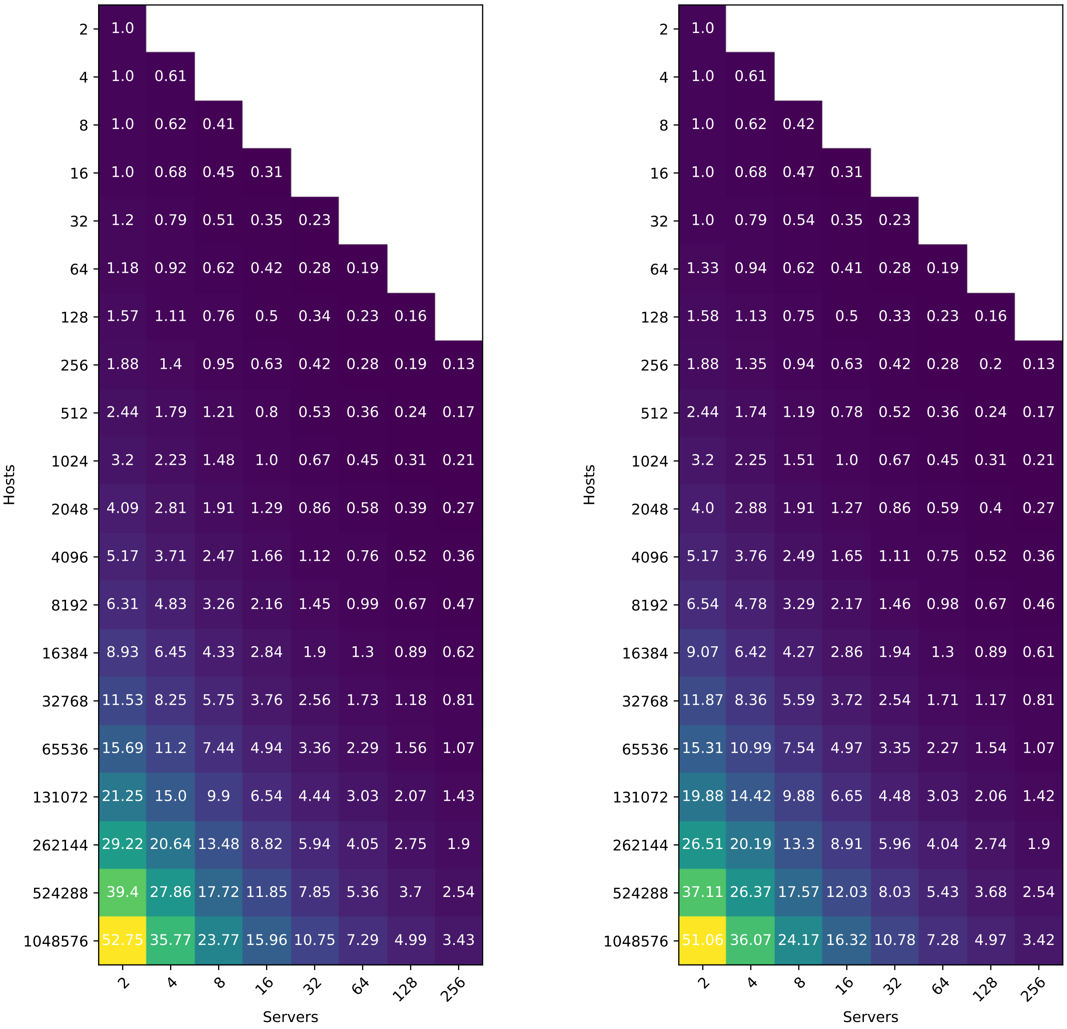
E.2.1 IPv4

In this section we expose the main polynomials tested on IPv4 addresses.



(a) CRC-16

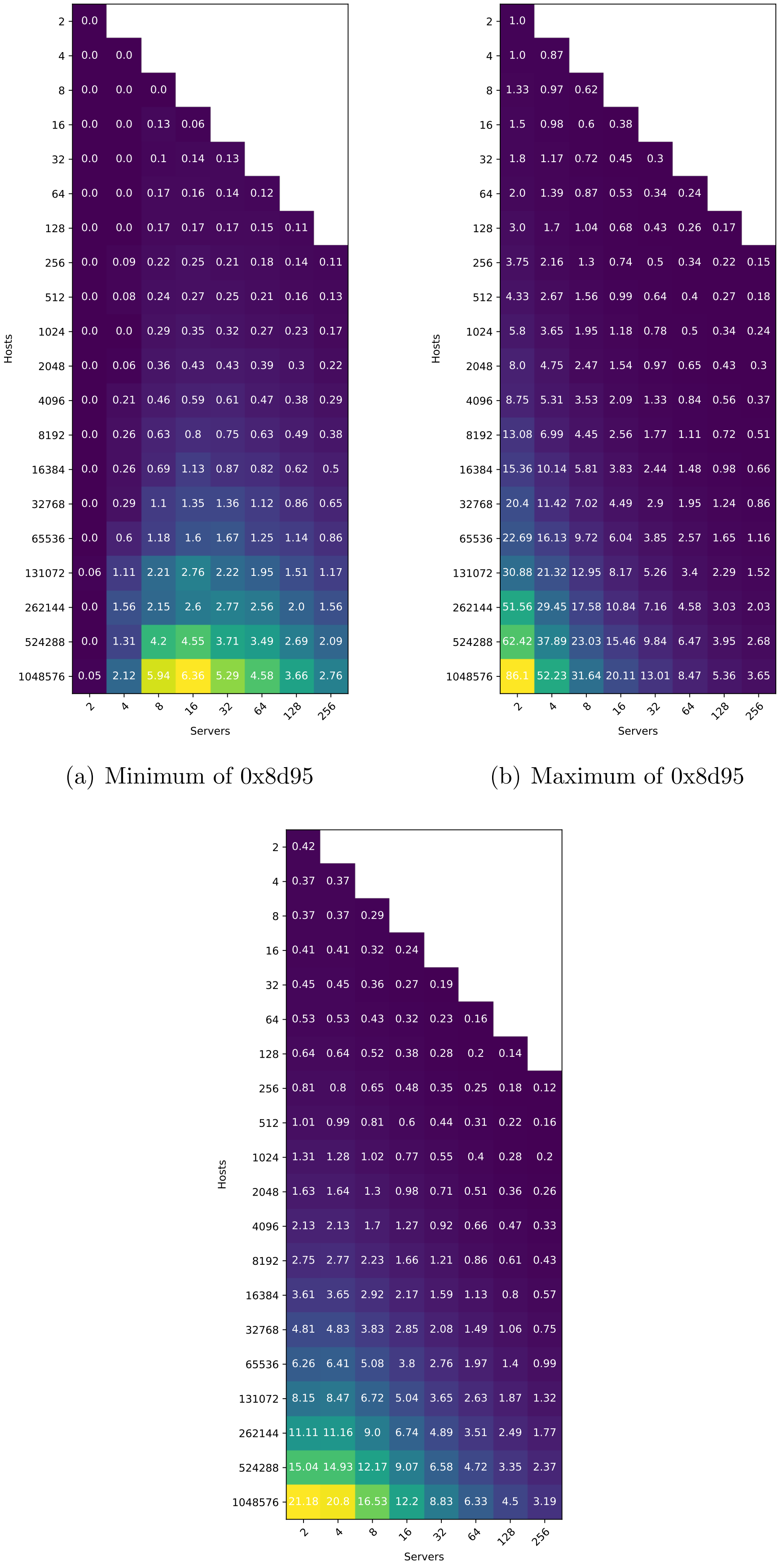
(b) CRC-32



(a) 0x8d95

(b) 0x973afb51

Figure E.2: IPv4 95 percentile of Mean Square Error for different polynomials



	2	4	8	16	32	64	128	256
2	0.0							
4	0.0	0.0						
8	0.0	0.0	0.0					
16	0.0	0.0	0.13	0.06				
32	0.0	0.0	0.1	0.14	0.13			
64	0.0	0.0	0.17	0.16	0.14	0.12		
128	0.0	0.0	0.17	0.17	0.17	0.15	0.11	
256	0.0	0.09	0.22	0.25	0.21	0.18	0.14	0.11
512	0.0	0.08	0.24	0.27	0.25	0.21	0.16	0.13
1024	0.0	0.0	0.29	0.35	0.32	0.27	0.23	0.17
2048	0.0	0.06	0.36	0.43	0.43	0.39	0.3	0.22
4096	0.0	0.21	0.46	0.59	0.61	0.47	0.38	0.29
8192	0.0	0.26	0.63	0.8	0.75	0.63	0.49	0.38
16384	0.0	0.26	0.69	1.13	0.87	0.82	0.62	0.5
32768	0.0	0.29	1.1	1.35	1.36	1.12	0.86	0.65
65536	0.0	0.6	1.18	1.6	1.67	1.25	1.14	0.86
131072	0.06	1.11	2.21	2.76	2.22	1.95	1.51	1.17
262144	0.0	1.56	2.15	2.6	2.77	2.56	2.0	1.56
524288	0.0	1.31	4.2	4.55	3.71	3.49	2.69	2.09
1048576	0.05	2.12	5.94	6.36	5.29	4.58	3.66	2.76

(a) Minimum of 0x8d95

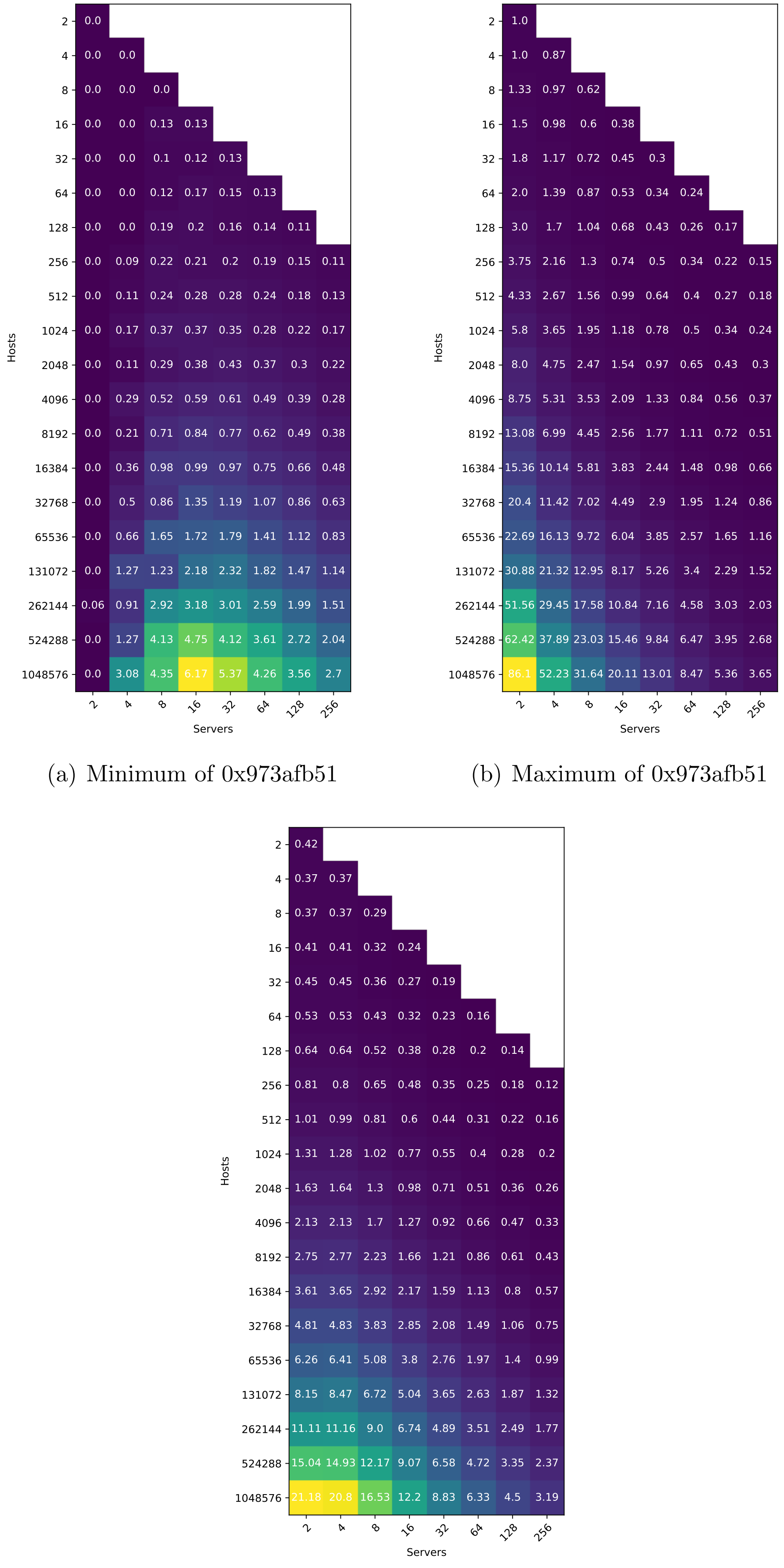
	2	4	8	16	32	64	128	256
2	1.0							
4	1.0	0.87						
8	1.33	0.97	0.62					
16	1.5	0.98	0.6	0.38				
32	1.8	1.17	0.72	0.45	0.3			
64	2.0	1.39	0.87	0.53	0.34	0.24		
128	3.0	1.7	1.04	0.68	0.43	0.26	0.17	
256	3.75	2.16	1.3	0.74	0.5	0.34	0.22	0.15
512	4.33	2.67	1.56	0.99	0.64	0.4	0.27	0.18
1024	5.8	3.65	1.95	1.18	0.78	0.5	0.34	0.24
2048	8.0	4.75	2.47	1.54	0.97	0.65	0.43	0.3
4096	8.75	5.31	3.53	2.09	1.33	0.84	0.56	0.37
8192	13.08	6.99	4.45	2.56	1.77	1.11	0.72	0.51
16384	15.36	10.14	5.81	3.83	2.44	1.48	0.98	0.66
32768	20.4	11.42	7.02	4.49	2.9	1.95	1.24	0.86
65536	22.69	16.13	9.72	6.04	3.85	2.57	1.65	1.16
131072	30.88	21.32	12.95	8.17	5.26	3.4	2.29	1.52
262144	51.56	29.45	17.58	10.84	7.16	4.58	3.03	2.03
524288	62.42	37.89	23.03	15.46	9.84	6.47	3.95	2.68
1048576	86.1	52.23	31.64	20.11	13.01	8.47	5.36	3.65

(b) Maximum of 0x8d95

	2	4	8	16	32	64	128	256
2	0.42							
4	0.37	0.37						
8	0.37	0.37	0.29					
16	0.41	0.41	0.32	0.24				
32	0.45	0.45	0.36	0.27	0.19			
64	0.53	0.53	0.43	0.32	0.23	0.16		
128	0.64	0.64	0.52	0.38	0.28	0.2	0.14	
256	0.81	0.8	0.65	0.48	0.35	0.25	0.18	0.12
512	1.01	0.99	0.81	0.6	0.44	0.31	0.22	0.16
1024	1.31	1.28	1.02	0.77	0.55	0.4	0.28	0.2
2048	1.63	1.64	1.3	0.98	0.71	0.51	0.36	0.26
4096	2.13	2.13	1.7	1.27	0.92	0.66	0.47	0.33
8192	2.75	2.77	2.23	1.66	1.21	0.86	0.61	0.43
16384	3.61	3.65	2.92	2.17	1.59	1.13	0.8	0.57
32768	4.81	4.83	3.83	2.85	2.08	1.49	1.06	0.75
65536	6.26	6.41	5.08	3.8	2.76	1.97	1.4	0.99
131072	8.15	8.47	6.72	5.04	3.65	2.63	1.87	1.32
262144	11.11	11.16	9.0	6.74	4.89	3.51	2.49	1.77
524288	15.04	14.93	12.17	9.07	6.58	4.72	3.35	2.37
1048576	21.18	20.8	16.53	12.2	8.83	6.33	4.5	3.19

(c) Average of 0x8d95

Figure E.3: IPv4 0x8d95 load balancing analysis



(a) Minimum of 0x973afb51

Hosts	2	4	8	16	32	64	128	256
2	0.0							
4	0.0	0.0						
8	0.0	0.0	0.0					
16	0.0	0.0	0.13	0.13				
32	0.0	0.0	0.1	0.12	0.13			
64	0.0	0.0	0.12	0.17	0.15	0.13		
128	0.0	0.0	0.19	0.2	0.16	0.14	0.11	
256	0.0	0.09	0.22	0.21	0.2	0.19	0.15	0.11
512	0.0	0.11	0.24	0.28	0.28	0.24	0.18	0.13
1024	0.0	0.17	0.37	0.37	0.35	0.28	0.22	0.17
2048	0.0	0.11	0.29	0.38	0.43	0.37	0.3	0.22
4096	0.0	0.29	0.52	0.59	0.61	0.49	0.39	0.28
8192	0.0	0.21	0.71	0.84	0.77	0.62	0.49	0.38
16384	0.0	0.36	0.98	0.99	0.97	0.75	0.66	0.48
32768	0.0	0.5	0.86	1.35	1.19	1.07	0.86	0.63
65536	0.0	0.66	1.65	1.72	1.79	1.41	1.12	0.83
131072	0.0	1.27	1.23	2.18	2.32	1.82	1.47	1.14
262144	0.06	0.91	2.92	3.18	3.01	2.59	1.99	1.51
524288	0.0	1.27	4.13	4.75	4.12	3.61	2.72	2.04
1048576	0.0	3.08	4.35	6.17	5.37	4.26	3.56	2.7

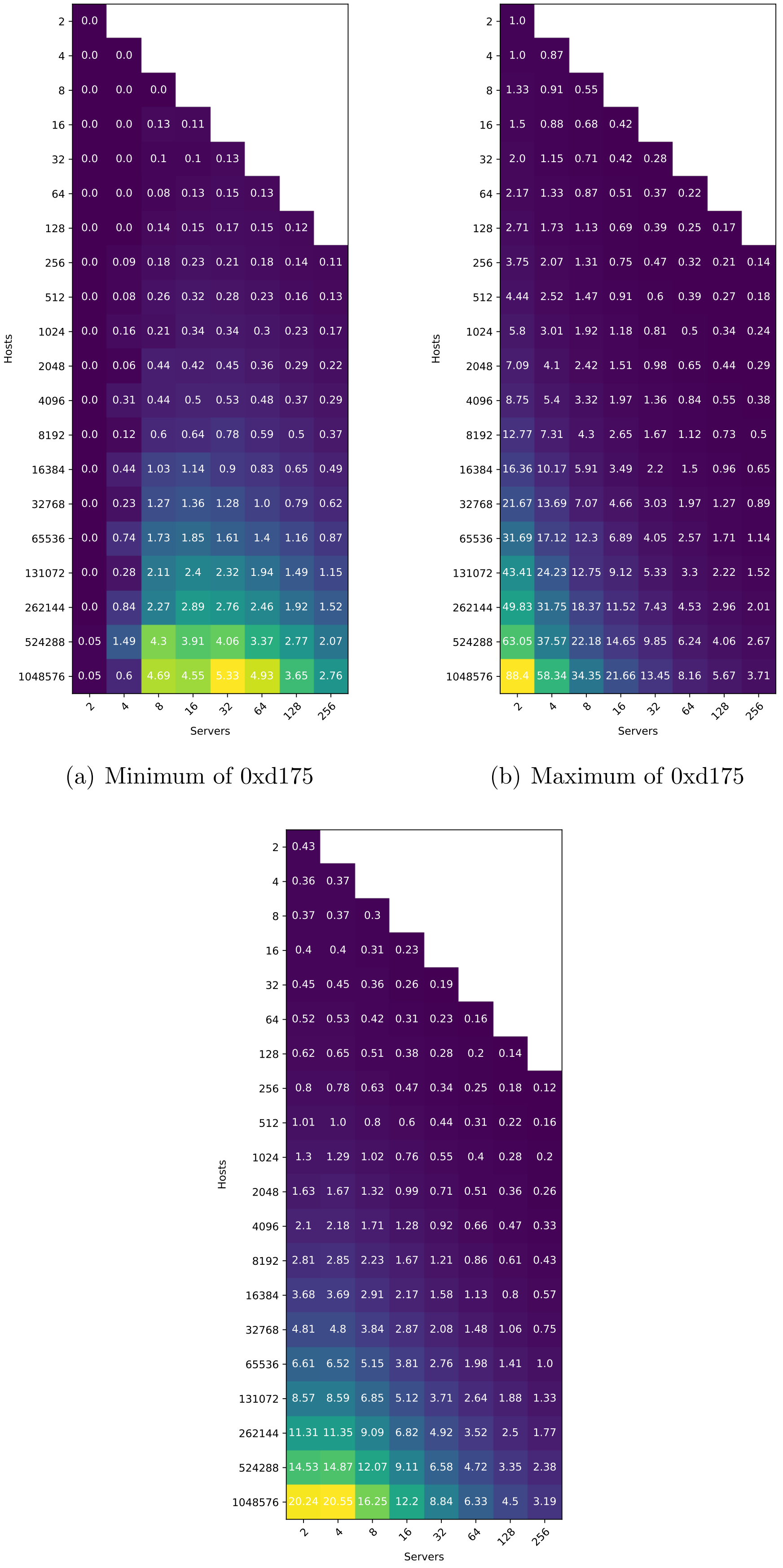
(b) Maximum of 0x973afb51

Hosts	2	4	8	16	32	64	128	256
2	1.0							
4	1.0	0.87						
8	1.33	0.97	0.62					
16	1.5	0.98	0.6	0.38				
32	1.8	1.17	0.72	0.45	0.3			
64	2.0	1.39	0.87	0.53	0.34	0.24		
128	3.0	1.7	1.04	0.68	0.43	0.26	0.17	
256	3.75	2.16	1.3	0.74	0.5	0.34	0.22	0.15
512	4.33	2.67	1.56	0.99	0.64	0.4	0.27	0.18
1024	5.8	3.65	1.95	1.18	0.78	0.5	0.34	0.24
2048	8.0	4.75	2.47	1.54	0.97	0.65	0.43	0.3
4096	8.75	5.31	3.53	2.09	1.33	0.84	0.56	0.37
8192	13.08	6.99	4.45	2.56	1.77	1.11	0.72	0.51
16384	15.36	10.14	5.81	3.83	2.44	1.48	0.98	0.66
32768	20.4	11.42	7.02	4.49	2.9	1.95	1.24	0.86
65536	22.69	16.13	9.72	6.04	3.85	2.57	1.65	1.16
131072	30.88	21.32	12.95	8.17	5.26	3.4	2.29	1.52
262144	51.56	29.45	17.58	10.84	7.16	4.58	3.03	2.03
524288	62.42	37.89	23.03	15.46	9.84	6.47	3.95	2.68
1048576	86.1	52.23	31.64	20.11	13.01	8.47	5.36	3.65

(c) Average of 0x973afb51

Hosts	2	4	8	16	32	64	128	256
2	0.42							
4	0.37	0.37						
8	0.37	0.37	0.29					
16	0.41	0.41	0.32	0.24				
32	0.45	0.45	0.36	0.27	0.19			
64	0.53	0.53	0.43	0.32	0.23	0.16		
128	0.64	0.64	0.52	0.38	0.28	0.2	0.14	
256	0.81	0.8	0.65	0.48	0.35	0.25	0.18	0.12
512	1.01	0.99	0.81	0.6	0.44	0.31	0.22	0.16
1024	1.31	1.28	1.02	0.77	0.55	0.4	0.28	0.2
2048	1.63	1.64	1.3	0.98	0.71	0.51	0.36	0.26
4096	2.13	2.13	1.7	1.27	0.92	0.66	0.47	0.33
8192	2.75	2.77	2.23	1.66	1.21	0.86	0.61	0.43
16384	3.61	3.65	2.92	2.17	1.59	1.13	0.8	0.57
32768	4.81	4.83	3.83	2.85	2.08	1.49	1.06	0.75
65536	6.26	6.41	5.08	3.8	2.76	1.97	1.4	0.99
131072	8.15	8.47	6.72	5.04	3.65	2.63	1.87	1.32
262144	11.11	11.16	9.0	6.74	4.89	3.51	2.49	1.77
524288	15.04	14.93	12.17	9.07	6.58	4.72	3.35	2.37
1048576	21.18	20.8	16.53	12.2	8.83	6.33	4.5	3.19

Figure E.4: IPv4 0x973afb51 load balancing analysis



Hosts	2	4	8	16	32	64	128	256
2	0.0							
4	0.0	0.0						
8	0.0	0.0	0.0					
16	0.0	0.0	0.13	0.11				
32	0.0	0.0	0.1	0.1	0.13			
64	0.0	0.0	0.08	0.13	0.15	0.13		
128	0.0	0.0	0.14	0.15	0.17	0.15	0.12	
256	0.0	0.09	0.18	0.23	0.21	0.18	0.14	0.11
512	0.0	0.08	0.26	0.32	0.28	0.23	0.16	0.13
1024	0.0	0.16	0.21	0.34	0.34	0.3	0.23	0.17
2048	0.0	0.06	0.44	0.42	0.45	0.36	0.29	0.22
4096	0.0	0.31	0.44	0.5	0.53	0.48	0.37	0.29
8192	0.0	0.12	0.6	0.64	0.78	0.59	0.5	0.37
16384	0.0	0.44	1.03	1.14	0.9	0.83	0.65	0.49
32768	0.0	0.23	1.27	1.36	1.28	1.0	0.79	0.62
65536	0.0	0.74	1.73	1.85	1.61	1.4	1.16	0.87
131072	0.0	0.28	2.11	2.4	2.32	1.94	1.49	1.15
262144	0.0	0.84	2.27	2.89	2.76	2.46	1.92	1.52
524288	0.05	1.49	4.3	3.91	4.06	3.37	2.77	2.07
1048576	0.05	0.6	4.69	4.55	5.33	4.93	3.65	2.76

(a) Minimum of 0xd175

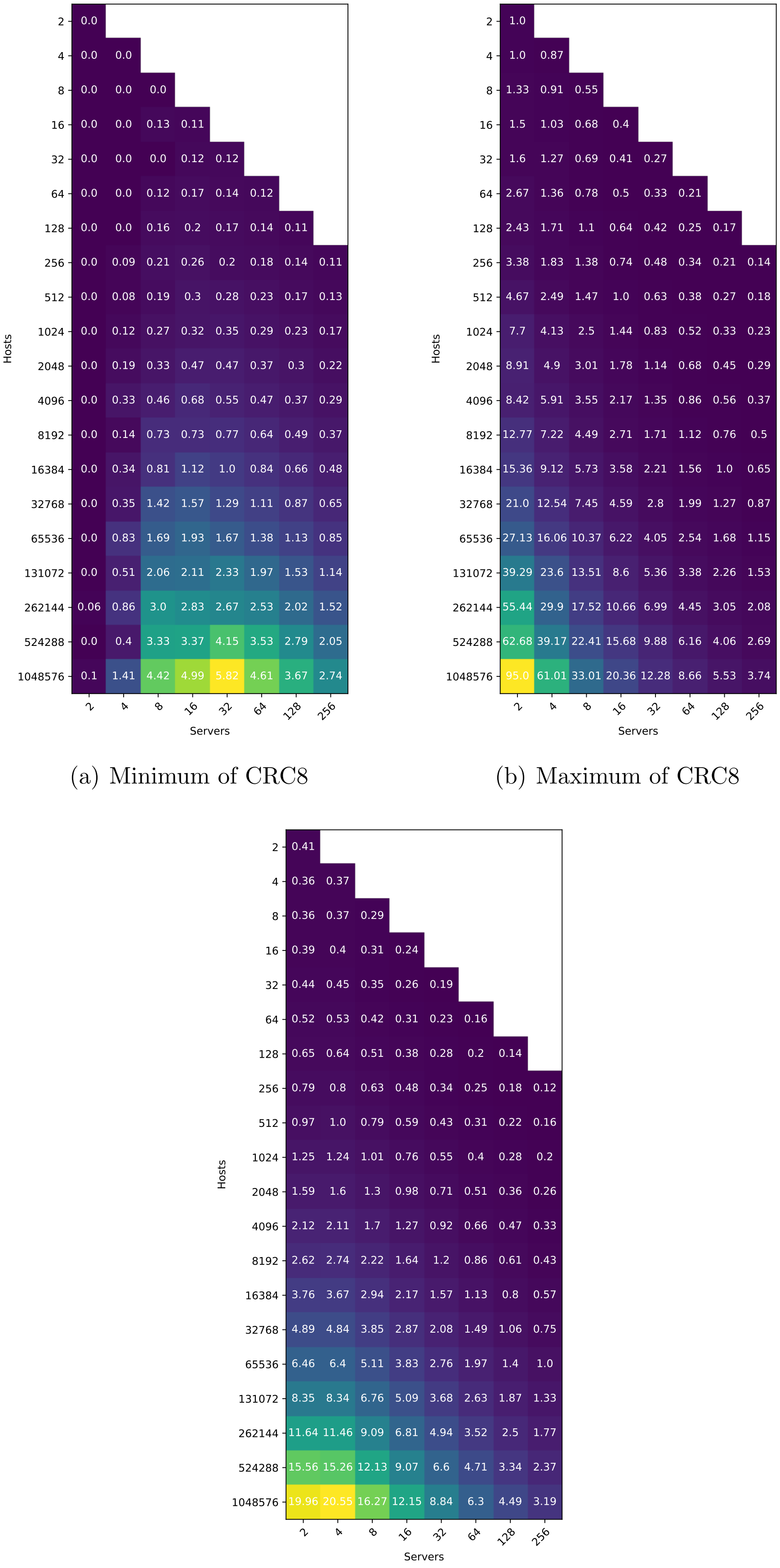
Hosts	2	4	8	16	32	64	128	256
2	1.0							
4	1.0	0.87						
8	1.33	0.91	0.55					
16	1.5	0.88	0.68	0.42				
32	2.0	1.15	0.71	0.42	0.28			
64	2.17	1.33	0.87	0.51	0.37	0.22		
128	2.71	1.73	1.13	0.69	0.39	0.25	0.17	
256	3.75	2.07	1.31	0.75	0.47	0.32	0.21	0.14
512	4.44	2.52	1.47	0.91	0.6	0.39	0.27	0.18
1024	5.8	3.01	1.92	1.18	0.81	0.5	0.34	0.24
2048	7.09	4.1	2.42	1.51	0.98	0.65	0.44	0.29
4096	8.75	5.4	3.32	1.97	1.36	0.84	0.55	0.38
8192	12.77	7.31	4.3	2.65	1.67	1.12	0.73	0.5
16384	16.36	10.17	5.91	3.49	2.2	1.5	0.96	0.65
32768	21.67	13.69	7.07	4.66	3.03	1.97	1.27	0.89
65536	31.69	17.12	12.3	6.89	4.05	2.57	1.71	1.14
131072	43.41	24.23	12.75	9.12	5.33	3.3	2.22	1.52
262144	49.83	31.75	18.37	11.52	7.43	4.53	2.96	2.01
524288	63.05	37.57	22.18	14.65	9.85	6.24	4.06	2.67
1048576	88.4	58.34	34.35	21.66	13.45	8.16	5.67	3.71

(b) Maximum of 0xd175

Hosts	2	4	8	16	32	64	128	256
2	0.43							
4	0.36	0.37						
8	0.37	0.37	0.3					
16	0.4	0.4	0.31	0.23				
32	0.45	0.45	0.36	0.26	0.19			
64	0.52	0.53	0.42	0.31	0.23	0.16		
128	0.62	0.65	0.51	0.38	0.28	0.2	0.14	
256	0.8	0.78	0.63	0.47	0.34	0.25	0.18	0.12
512	1.01	1.0	0.8	0.6	0.44	0.31	0.22	0.16
1024	1.3	1.29	1.02	0.76	0.55	0.4	0.28	0.2
2048	1.63	1.67	1.32	0.99	0.71	0.51	0.36	0.26
4096	2.1	2.18	1.71	1.28	0.92	0.66	0.47	0.33
8192	2.81	2.85	2.23	1.67	1.21	0.86	0.61	0.43
16384	3.68	3.69	2.91	2.17	1.58	1.13	0.8	0.57
32768	4.81	4.8	3.84	2.87	2.08	1.48	1.06	0.75
65536	6.61	6.52	5.15	3.81	2.76	1.98	1.41	1.0
131072	8.57	8.59	6.85	5.12	3.71	2.64	1.88	1.33
262144	11.31	11.35	9.09	6.82	4.92	3.52	2.5	1.77
524288	14.53	14.87	12.07	9.11	6.58	4.72	3.35	2.38
1048576	20.24	20.55	16.25	12.2	8.84	6.33	4.5	3.19

(c) Average of 0xd175

Figure E.5: IPv4 0xd175 load balancing analysis

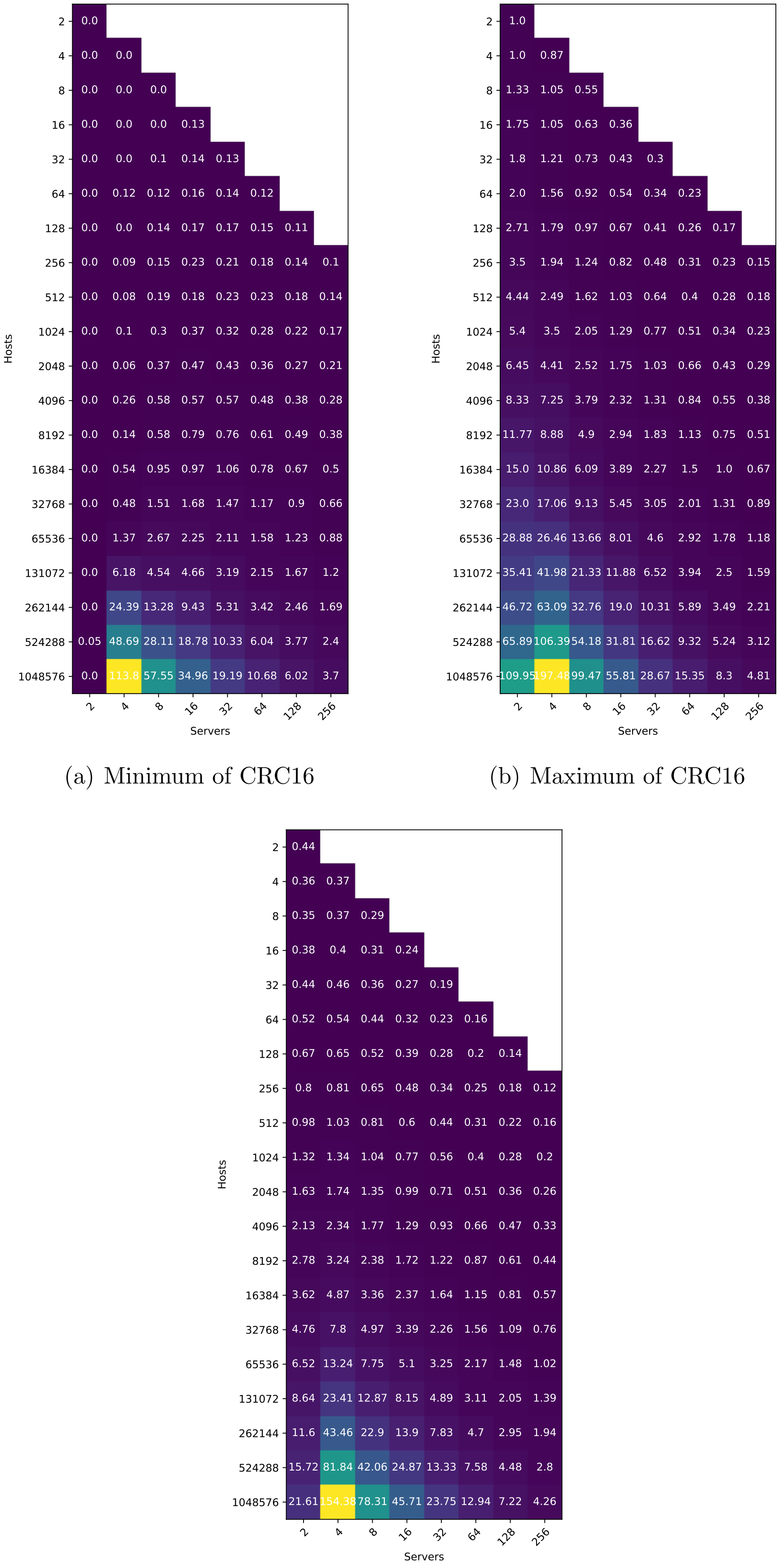


(a) Minimum of CRC8

(b) Maximum of CRC8

(c) Average of CRC8

Figure E.6: IPv4 CRC8 load balancing analysis

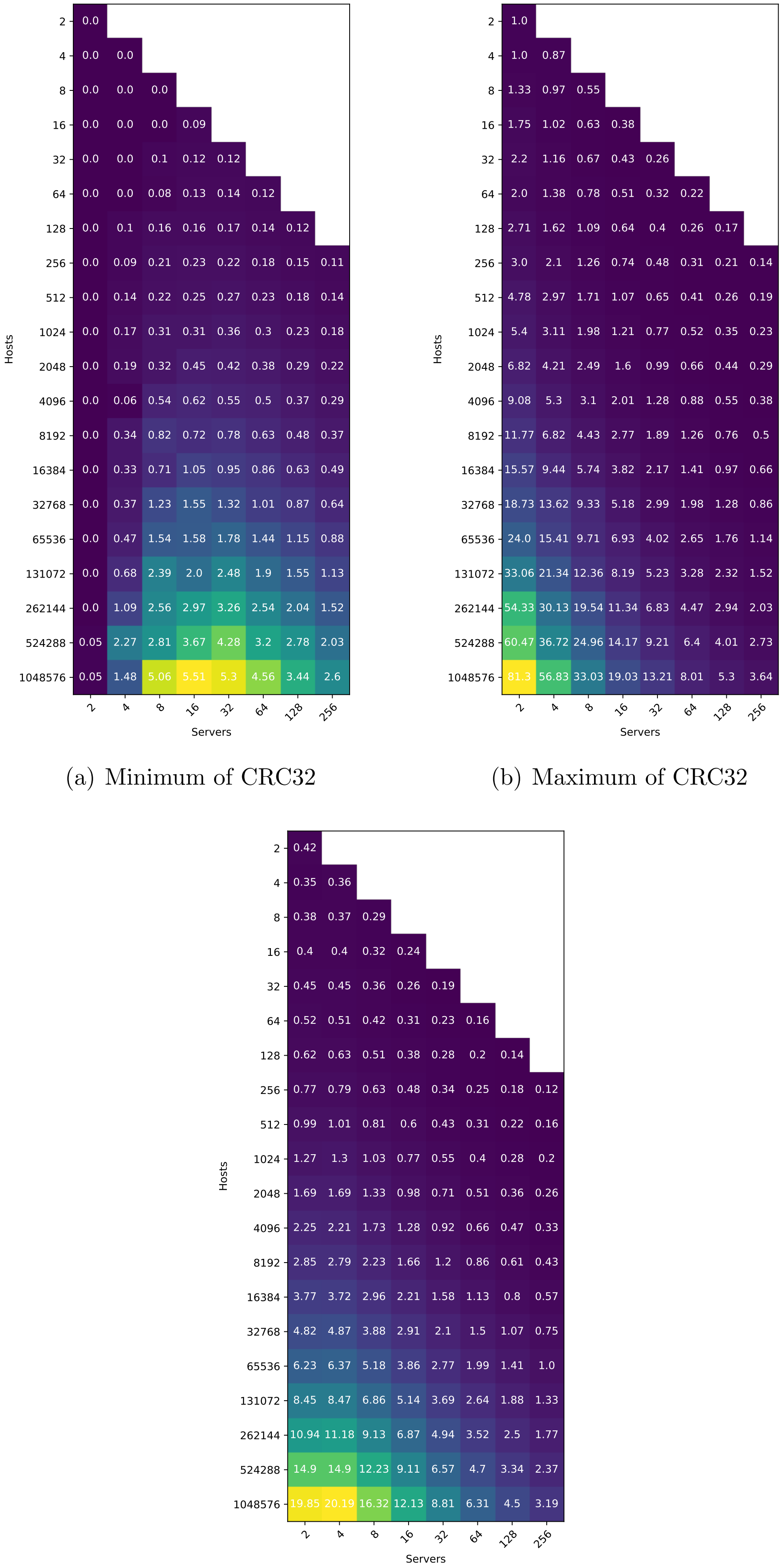


(a) Minimum of CRC16

(b) Maximum of CRC16

(c) Average of CRC16

Figure E.7: IPv4 CRC16 load balancing analysis

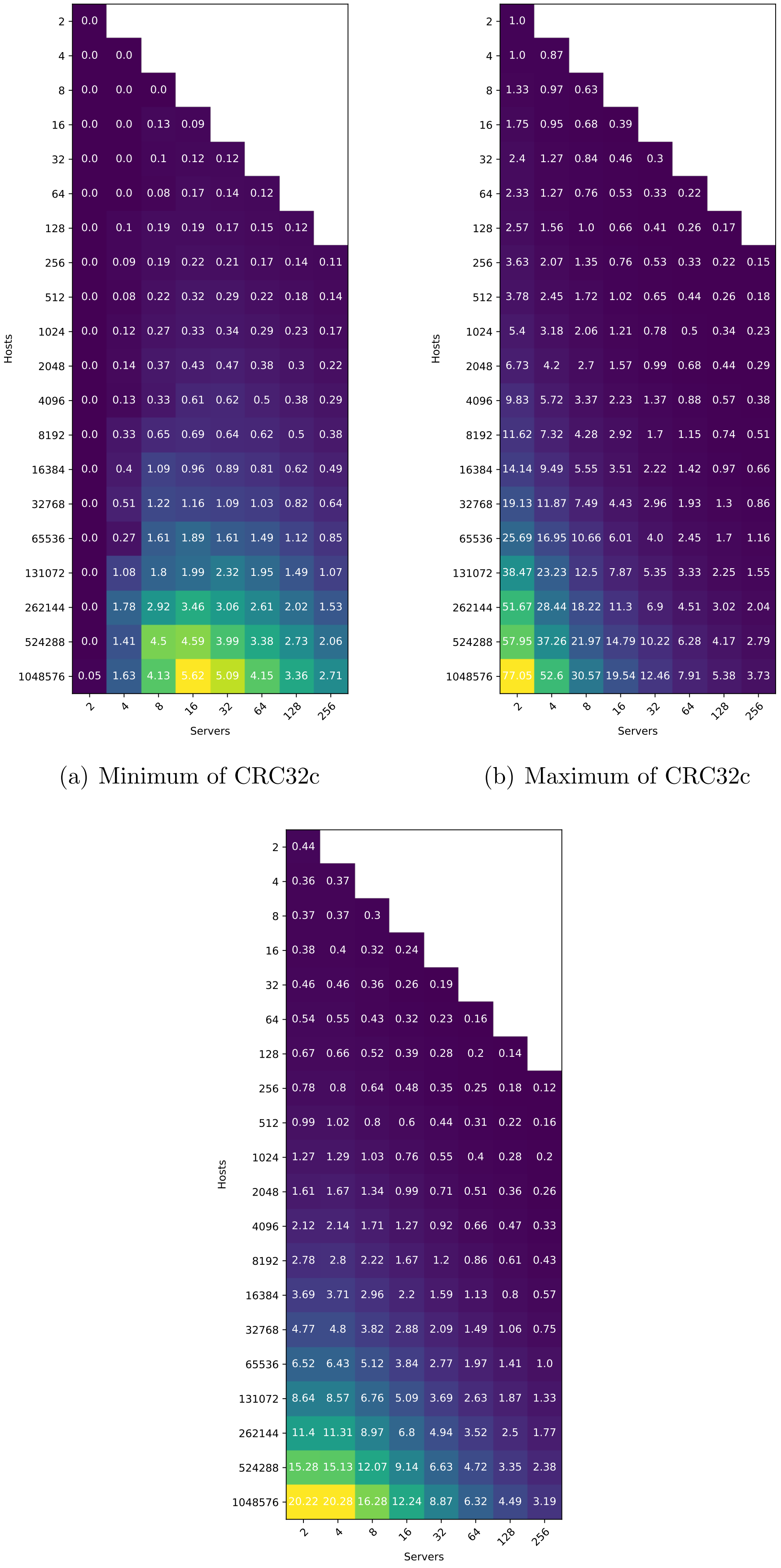


(a) Minimum of CRC32

(b) Maximum of CRC32

(c) Average of CRC32

Figure E.8: IPv4 CRC32 load balancing analysis



(a) Minimum of CRC32c

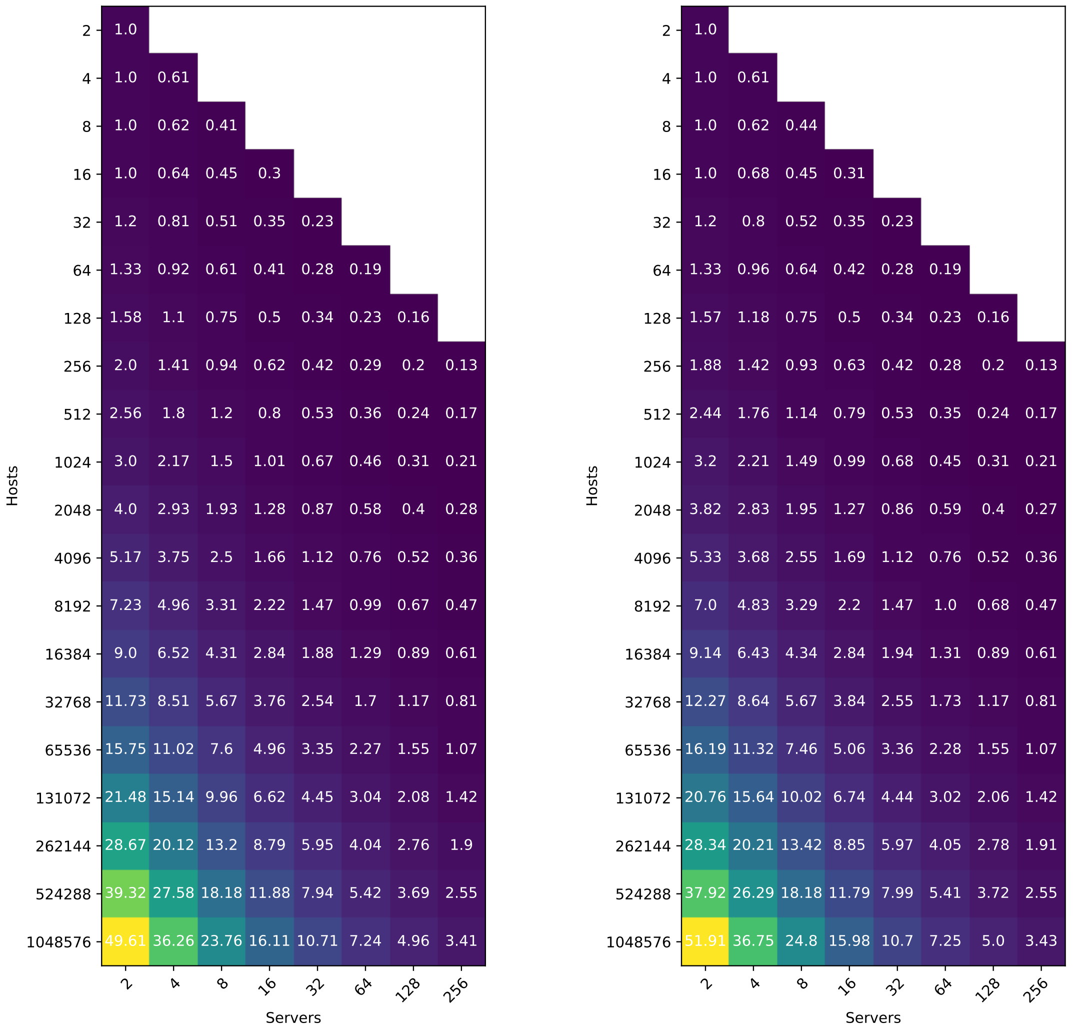
(b) Maximum of CRC32c

(c) Average of CRC32c

Figure E.9: IPv4 CRC32c load balancing analysis

E.2.2 IPv6

In this section we expose the main polynomials tested on IPv6 addresses.



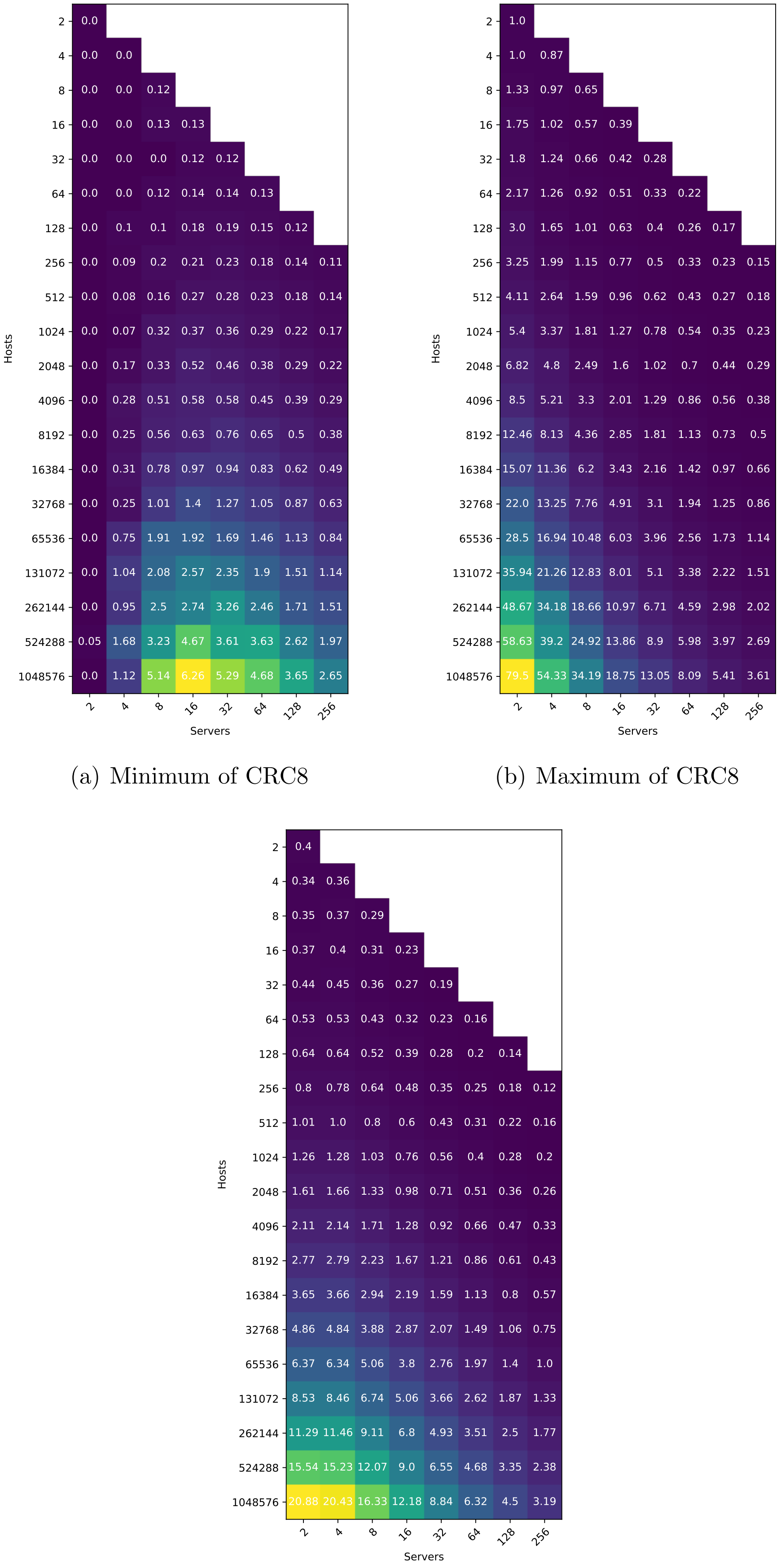
(a) CRC-16

Hosts	2	4	8	16	32	64	128	256
2	1.0							
4	1.0	0.61						
8	1.0	0.62	0.41					
16	1.0	0.64	0.45	0.3				
32	1.2	0.81	0.51	0.35	0.23			
64	1.33	0.92	0.61	0.41	0.28	0.19		
128	1.58	1.1	0.75	0.5	0.34	0.23	0.16	
256	2.0	1.41	0.94	0.62	0.42	0.29	0.2	0.13
512	2.56	1.8	1.2	0.8	0.53	0.36	0.24	0.17
1024	3.0	2.17	1.5	1.01	0.67	0.46	0.31	0.21
2048	4.0	2.93	1.93	1.28	0.87	0.58	0.4	0.28
4096	5.17	3.75	2.5	1.66	1.12	0.76	0.52	0.36
8192	7.23	4.96	3.31	2.22	1.47	0.99	0.67	0.47
16384	9.0	6.52	4.31	2.84	1.88	1.29	0.89	0.61
32768	11.73	8.51	5.67	3.76	2.54	1.7	1.17	0.81
65536	15.75	11.02	7.6	4.96	3.35	2.27	1.55	1.07
131072	21.48	15.14	9.96	6.62	4.45	3.04	2.08	1.42
262144	28.67	20.12	13.2	8.79	5.95	4.04	2.76	1.9
524288	39.32	27.58	18.18	11.88	7.94	5.42	3.69	2.55
1048576	49.61	36.26	23.76	16.11	10.71	7.24	4.96	3.41

(b) CRC-32

Hosts	2	4	8	16	32	64	128	256
2	1.0							
4	1.0	0.61						
8	1.0	0.62	0.44					
16	1.0	0.68	0.45	0.31				
32	1.2	0.8	0.52	0.35	0.23			
64	1.33	0.96	0.64	0.42	0.28	0.19		
128	1.57	1.18	0.75	0.5	0.34	0.23	0.16	
256	1.88	1.42	0.93	0.63	0.42	0.28	0.2	0.13
512	2.44	1.76	1.14	0.79	0.53	0.35	0.24	0.17
1024	3.2	2.21	1.49	0.99	0.68	0.45	0.31	0.21
2048	3.82	2.83	1.95	1.27	0.86	0.59	0.4	0.27
4096	5.33	3.68	2.55	1.69	1.12	0.76	0.52	0.36
8192	7.0	4.83	3.29	2.2	1.47	1.0	0.68	0.47
16384	9.14	6.43	4.34	2.84	1.94	1.31	0.89	0.61
32768	12.27	8.64	5.67	3.84	2.55	1.73	1.17	0.81
65536	16.19	11.32	7.46	5.06	3.36	2.28	1.55	1.07
131072	20.76	15.64	10.02	6.74	4.44	3.02	2.06	1.42
262144	28.34	20.21	13.42	8.85	5.97	4.05	2.78	1.91
524288	37.92	26.29	18.18	11.79	7.99	5.41	3.72	2.55
1048576	51.91	36.75	24.8	15.98	10.7	7.25	5.0	3.43

Figure E.10: IPv6 95 percentile of Mean Square Error for different polynomials

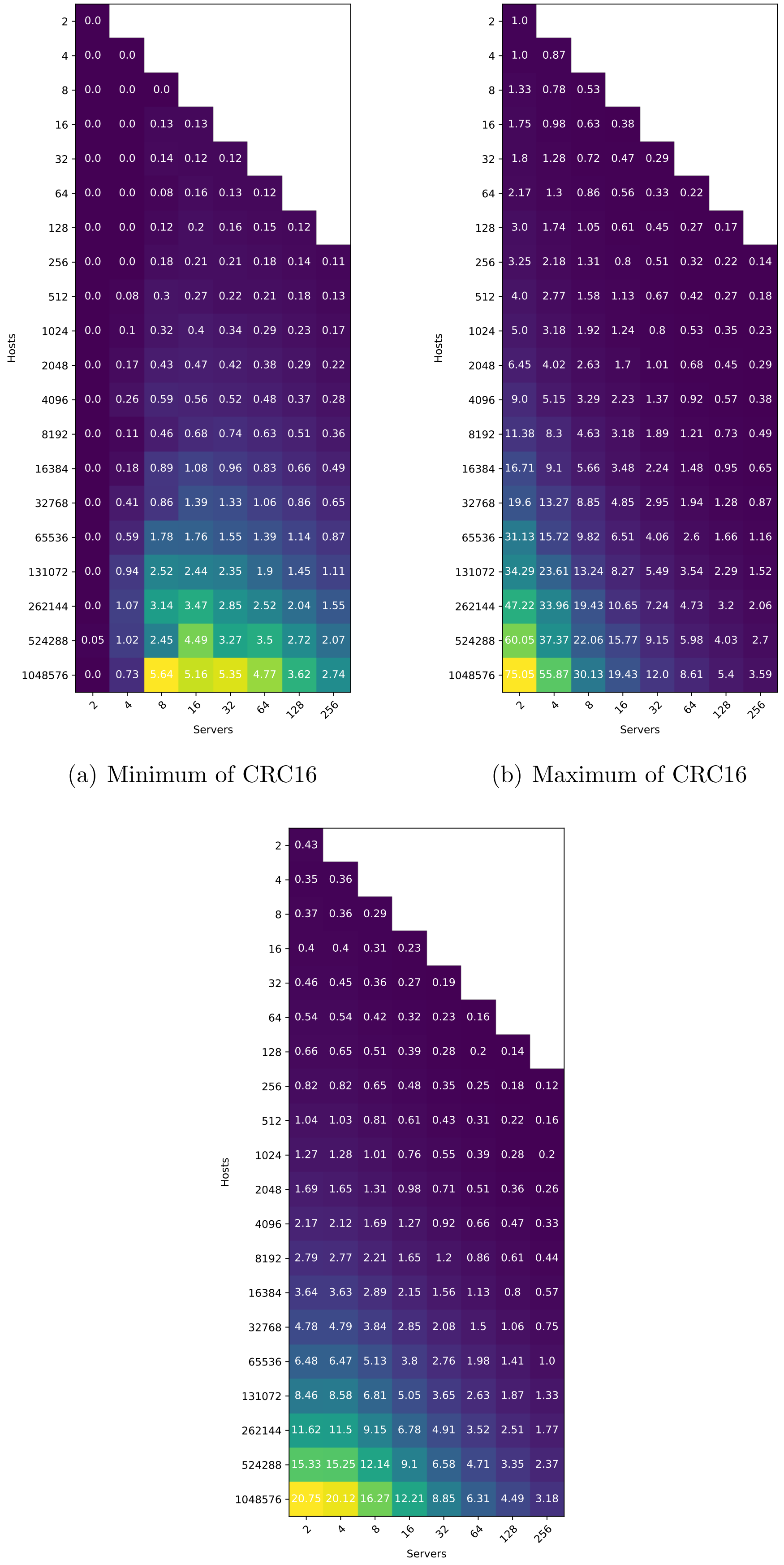


(a) Minimum of CRC8

(b) Maximum of CRC8

(c) Average of CRC8

Figure E.11: IPv6 CRC8 load balancing analysis

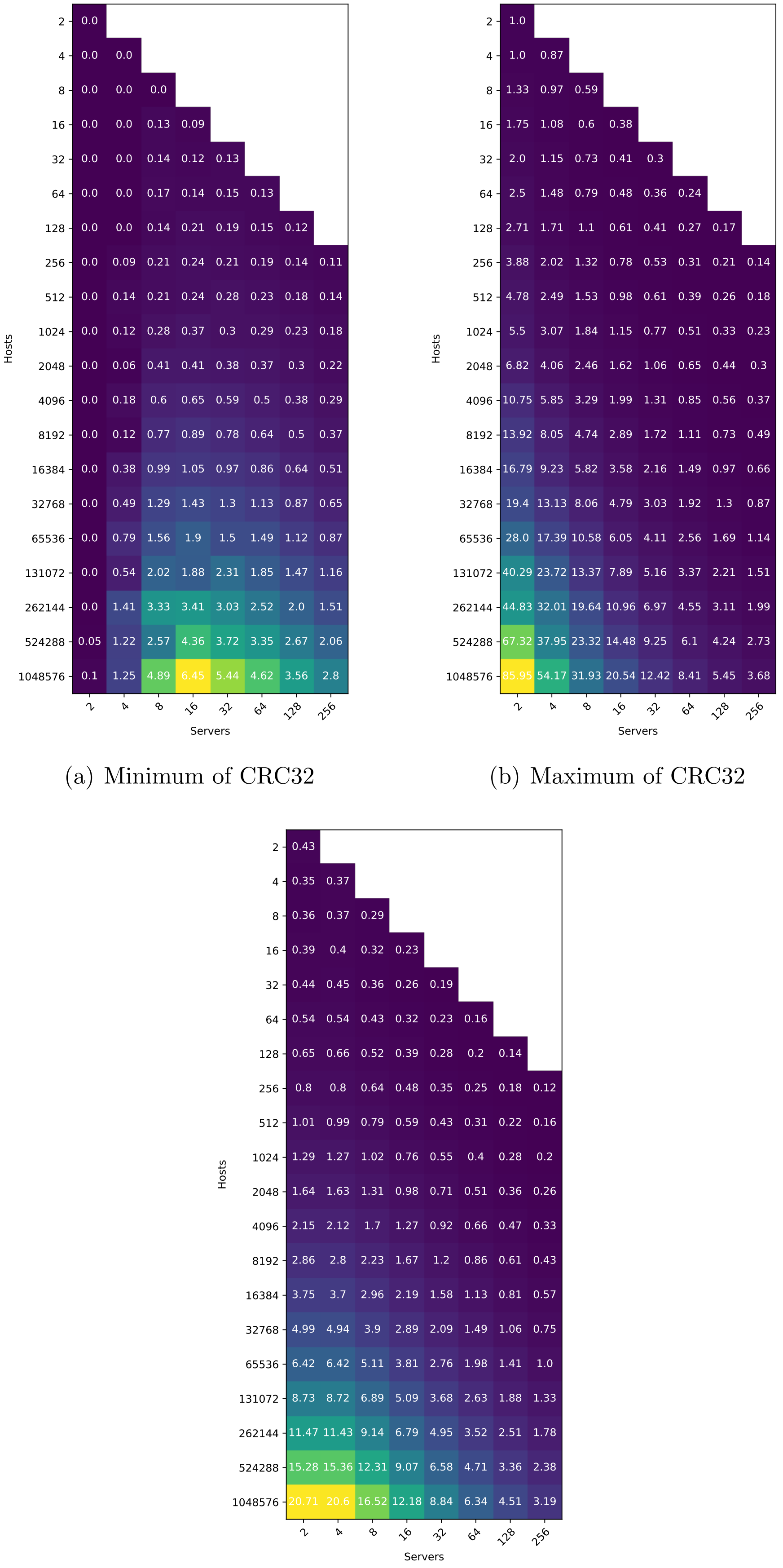


(a) Minimum of CRC16

(b) Maximum of CRC16

(c) Average of CRC16

Figure E.12: IPv6 CRC16 load balancing analysis



	2	4	8	16	32	64	128	256
2	0.0							
4	0.0	0.0						
8	0.0	0.0	0.0					
16	0.0	0.0	0.13	0.09				
32	0.0	0.0	0.14	0.12	0.13			
64	0.0	0.0	0.17	0.14	0.15	0.13		
128	0.0	0.0	0.14	0.21	0.19	0.15	0.12	
256	0.0	0.09	0.21	0.24	0.21	0.19	0.14	0.11
512	0.0	0.14	0.21	0.24	0.28	0.23	0.18	0.14
1024	0.0	0.12	0.28	0.37	0.3	0.29	0.23	0.18
2048	0.0	0.06	0.41	0.41	0.38	0.37	0.3	0.22
4096	0.0	0.18	0.6	0.65	0.59	0.5	0.38	0.29
8192	0.0	0.12	0.77	0.89	0.78	0.64	0.5	0.37
16384	0.0	0.38	0.99	1.05	0.97	0.86	0.64	0.51
32768	0.0	0.49	1.29	1.43	1.3	1.13	0.87	0.65
65536	0.0	0.79	1.56	1.9	1.5	1.49	1.12	0.87
131072	0.0	0.54	2.02	1.88	2.31	1.85	1.47	1.16
262144	0.0	1.41	3.33	3.41	3.03	2.52	2.0	1.51
524288	0.05	1.22	2.57	4.36	3.72	3.35	2.67	2.06
1048576	0.1	1.25	4.89	6.45	5.44	4.62	3.56	2.8

(a) Minimum of CRC32

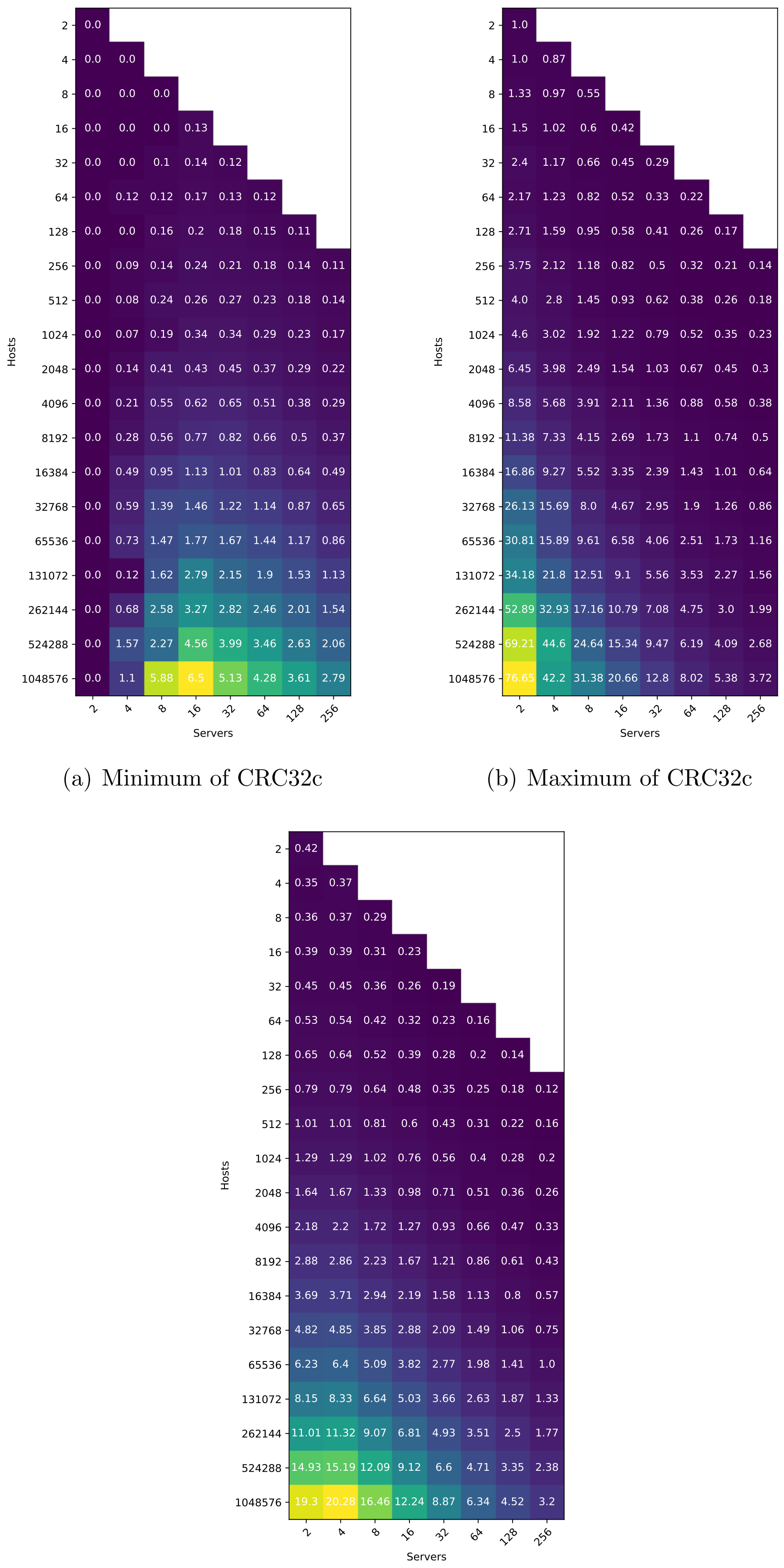
	2	4	8	16	32	64	128	256
2	1.0							
4	1.0	0.87						
8	1.33	0.97	0.59					
16	1.75	1.08	0.6	0.38				
32	2.0	1.15	0.73	0.41	0.3			
64	2.5	1.48	0.79	0.48	0.36	0.24		
128	2.71	1.71	1.1	0.61	0.41	0.27	0.17	
256	3.88	2.02	1.32	0.78	0.53	0.31	0.21	0.14
512	4.78	2.49	1.53	0.98	0.61	0.39	0.26	0.18
1024	5.5	3.07	1.84	1.15	0.77	0.51	0.33	0.23
2048	6.82	4.06	2.46	1.62	1.06	0.65	0.44	0.3
4096	10.75	5.85	3.29	1.99	1.31	0.85	0.56	0.37
8192	13.92	8.05	4.74	2.89	1.72	1.11	0.73	0.49
16384	16.79	9.23	5.82	3.58	2.16	1.49	0.97	0.66
32768	19.4	13.13	8.06	4.79	3.03	1.92	1.3	0.87
65536	28.0	17.39	10.58	6.05	4.11	2.56	1.69	1.14
131072	40.29	23.72	13.37	7.89	5.16	3.37	2.21	1.51
262144	44.83	32.01	19.64	10.96	6.97	4.55	3.11	1.99
524288	67.32	37.95	23.32	14.48	9.25	6.1	4.24	2.73
1048576	85.95	54.17	31.93	20.54	12.42	8.41	5.45	3.68

(b) Maximum of CRC32

	2	4	8	16	32	64	128	256
2	0.43							
4	0.35	0.37						
8	0.36	0.37	0.29					
16	0.39	0.4	0.32	0.23				
32	0.44	0.45	0.36	0.26	0.19			
64	0.54	0.54	0.43	0.32	0.23	0.16		
128	0.65	0.66	0.52	0.39	0.28	0.2	0.14	
256	0.8	0.8	0.64	0.48	0.35	0.25	0.18	0.12
512	1.01	0.99	0.79	0.59	0.43	0.31	0.22	0.16
1024	1.29	1.27	1.02	0.76	0.55	0.4	0.28	0.2
2048	1.64	1.63	1.31	0.98	0.71	0.51	0.36	0.26
4096	2.15	2.12	1.7	1.27	0.92	0.66	0.47	0.33
8192	2.86	2.8	2.23	1.67	1.2	0.86	0.61	0.43
16384	3.75	3.7	2.96	2.19	1.58	1.13	0.81	0.57
32768	4.99	4.94	3.9	2.89	2.09	1.49	1.06	0.75
65536	6.42	6.42	5.11	3.81	2.76	1.98	1.41	1.0
131072	8.73	8.72	6.89	5.09	3.68	2.63	1.88	1.33
262144	11.47	11.43	9.14	6.79	4.95	3.52	2.51	1.78
524288	15.28	15.36	12.31	9.07	6.58	4.71	3.36	2.38
1048576	20.71	20.6	16.52	12.18	8.84	6.34	4.51	3.19

(c) Average of CRC32

Figure E.13: IPv6 CRC32 load balancing analysis

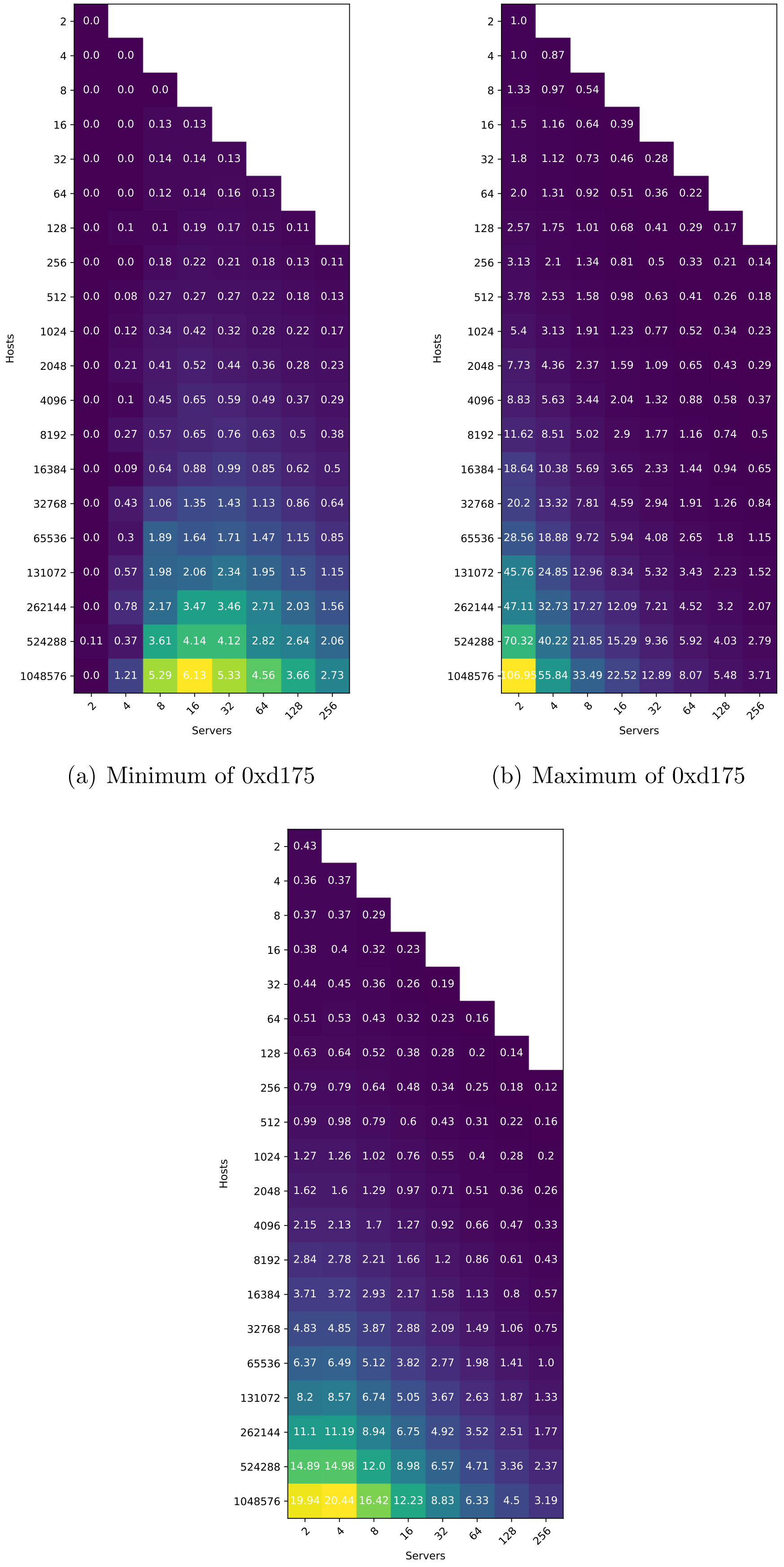


(a) Minimum of CRC32c

(b) Maximum of CRC32c

(c) Average of CRC32c

Figure E.14: IPv6 CRC32c load balancing analysis



(a) Minimum of 0xd175

(b) Maximum of 0xd175

(c) Average of 0xd175

Figure E.15: IPv6 0xd175 load balancing analysis