UNIVERSIDADE ESTADUAL DE CAMPINAS
Faculdade de Engenharia Elétrica e de Computação

Pattam Gyanesh Kumar Patra

MACSAD: Multi-Architecture Compiler System for Abstract Dataplanes

MACSAD: Sistema de Compilador Multi-Arquitetura para Planos de Dados Abstratos

Campinas

2019

Pattam Gyanesh Kumar Patra

MACSAD: Multi-Architecture Compiler System for Abstract Dataplanes

MACSAD: Sistema de Compilador Multi-Arquitetura para Planos de Dados Abstratos

Thesis presented to the Faculty of Electrical and Computer Engineering of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor, in the area of Computer Engineering.

Tese 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 Doutor em Engenharia Elétrica, na Área de Engenharia de Computação.

Supervisor: Prof. Dr. Christian Rodolfo Esteve Rothenberg

Este exemplar corresponde à versão final da tese defendida pelo aluno Pattam Gyanesh Kumar Patra, e orientada pelo Prof. Dr. Christian Rodolfo Esteve Rothenberg

---

Campinas

2019

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

P274m

Patra, Pattam Gyanesh Kumar, 1986-
MACSAD: Multi-Architecture Compiler System for Abstract Dataplanes / Pattam Gyanesh Kumar Patra. – Campinas, SP : [s.n.], 2019.

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

1. Redes definidas por software (Tecnologia de rede de computador). 2. Comutação de pacotes (Transmissão de dados). 3. Software - Desempenho. 4. Aprendizagem supervisionada (Aprendizado do computador). I. Esteve Rothenberg, Christian Rodolfo, 1982-. II. Universidade Estadual de Campinas. Faculdade de Engenharia Elétrica e de Computação. III. Título.

Informações para Biblioteca Digital

Título em outro idioma: MACSAD: Sistema de Compilador Multi-Arquitetura para Planos de Dados Abstratos
Palavras-chave em inglês:
Software-defined networking (Computer network technology)
Packet Switching (Data transmission)
Network performance
Supervised learning (Machine learning)
Área de concentração: Engenharia de Computação
Titulação: Doutor em Engenharia Elétrica
Banca examinadora:
Christian Rodolfo Esteve Rothenberg [Orientador]
Rodolfo Jardim de Azevedo
Leonardo de Souza Mendes
Rodolfo da Silva Villaça
Fernando Manuel Valente Ramos
Data de defesa: 21-05-2019
Programa de Pós-Graduação: Engenharia Elétrica

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-5106-4336
- Currículo Lattes do autor: http://lattes.cnpq.br/8044336523774815

COMISSÃO JULGADORA – TESE DE DOUTORADO

Candidato: Pattam Gyanesh Kumar Patra RA: 153806

Data da Defesa: 21 de Maio de 2019

Título da Tese: "MACSAD: Multi-Architecture Compiler System for Abstract Dataplanes".

Prof. Dr. Christian Rodolfo Esteve Rothenberg

Prof. Dr. Rodolfo Jardim de Azevedo

Prof. Dr. Leonardo de Souza Mendes

Prof. Dr. Rodolfo da Silva Villaca

Prof. Dr. Fernando Manuel Valente Ramos

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

Dedicated To

My family who gave me reason to smile, to cry, to get angry, who challenge me to reach beyond myself. My friend circle, JWALKERS, for being with me for the last 15 years, offered their shoulders to lean on, opened their arms to offer solace, stood by me to protect, and gave me a sense of existence, importance and aliveness everyday. A friend whom i met accidentally here in Brazil after 8 long years turning out to be a great thing as he continue to provide critical advises in every important juncture of my personal life & became the magic glue to keep both my personal and professional life sane. The amazing new friends i made in INTRIG who welcomed me to their hearts and to their home out of curiosity, love and respect, whom i now carry with me for life. Gergely Pongrácz for showing faith, for listening and for offering the valuable guidance i needed. Specially, to the stranger i put my complete faith on even before coming to Brazil, who became much more than a friend or an advisor, my YODA, Christian Rothenberg.

Acknowledgements

This work was supported by the Innovation Center, Ericsson Telecomunicações S.A., Brazil under grant agreement UNI.61 through Funcamp/Unicamp intermediation.

Cognizance of newth & vicissitude is all i long,
for it To live until I live.
(speaking for myself)

Abstract

Software-Defined Networking (SDN) strives for programmable data plane, yet flexible and scalable control and application planes. Despite having received less attention compared to control and application aspects of SDN, data planes are a critical piece of the SDN puzzle. We envision a flexible data plane showing characteristics, namely, *Programmability, Portability, Performance*, and *Scalability* (3PS) as different aspects of flexibility. While Programmability & Portability aspects focus on the architecture and design of the data plane, Performance & Scalability appears during the evaluation of it. We extend the focus of data plane evolution from Programmability from SDN school of thought to include Portability aspect of flexibility. Programmable data plane confirms to protocol-independent nature, whereas Portability addresses multi-architecture requirements of data plane design. P4 language, a new entrant, being a protocol-independent and target-independent high-level programming language is capable to take data plane evolution to the next level by unlocking the desired facets of data plane flexibility. To bring this required level of flexibility to a data plane, a multi-architecture compiler system is necessary which can compile a P4 program conforming to protocol & target independence nature of P4; However, such a unified compiler system solution is what we lack of. The main contribution of this thesis, the Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD) proposal, is an effort to fill the gap by extending the Top-Down approach of P4 towards programmability with Bottom-Up approach of OpenDataPlane (ODP) towards target-independence with its low-level but cross-platform (HW & SW) APIs. We strengthen the contributions of this thesis by including *Performance*, and *Scalability* aspects of flexibility too as part of our evaluation of MACSAD in multiple realistic scenarios.

Keywords: MACSAD; P4; OpenDataPlane; Software-Defined Networking; Programmable Dataplane; Performance analysis.

RESUMO

Redes Definidas por Software (Software-Defined Networking - SDN) almejam um plano de dados programável, além de planos de controle e aplicação flexíveis e escaláveis. Apesar de ter recebido menor atenção quando comparado aos aspectos dos planos de controle e aplicação, o plano de dados concerne uma peça chave nos enigmas de SDN. Nós contemplamos um plano de dados flexível apresentando as características, nomeadas, Programabilidade, Portabilidade, Desempenho e Escalabilidade (Programmability, Portability, Performance, and Scalability - 3PS) como diferentes aspectos de flexibilidade. Enquanto os aspectos de Programabilidade e Portabilidade focam na arquitetura e projeto do plano de dados, Desempenho e Escalabilidade aparecem durante a avaliação do mesmo. Estendemos o foco da evolução do plano de dados de Programabilidade da escola de pensamento SDN para incluir Portabilidade como aspecto de flexibilidade. O plano de dados programável confirma a natureza independente do protocolo, enquanto a Portabilidade atende aos requisitos de arquitetura múltipla do projeto do plano de dados. A linguagem P4, uma nova entrante, sendo uma linguagem de programação de alto nível independente do protocolo e independente do alvo, é capaz de levar a evolução do plano de dados ao próximo nível, desbloqueando as facetas desejadas da flexibilidade do plano de dados. Para trazer esse nível necessário de flexibilidade para um plano de dados, é necessário um sistema de compilador com várias arquiteturas que possa compilar um programa P4 em conformidade com o protocolo e a natureza de independência de destino de P4; No entanto, essa solução de sistema de compilador unificado é o que nos falta. A principal contribuição desta tese, a proposta do Sistema de Compiladores de Arquitetura Múltipla para Planos de Dados (Multi-Architecture Compiler System for Abstract Dataplanes - MACSAD), é um esforço para preencher a lacuna estendendo a abordagem Top-Down de P4 em direção à programabilidade com a abordagem Bottom-Up do OpenDataPlane (ODP) em direção à independência de destino com suas APIs de baixo nível, mas de plataforma cruzada (HW & SW). Reforçamos as contribuições desta tese incluindo aspectos de Desempenho e Escalabilidade da flexibilidade também como parte de nossa avaliação do MACSAD em múltiplos cenários realistas.

Palavras-chaves: MACSAD; P4; OpenDataPlane; Redes Definidas por Software; Plano de Dados Programável; Análise de Desempenho.

List of Figures

Figure 1 – Supported Features envisioned by MACSAD 26
Figure 2 – Where ODP is situated? 35
Figure 3 – PISA Architecture. Based on:(MCKEOWN, 2015) 38
Figure 4 – P4 Abstract Forwarding Model 40
Figure 5 – Components of a P4 Program 40
Figure 6 – P4 Parser Graph Example 43
Figure 7 – P4Runtime Reference Architecture 47
Figure 8 – High-level Reference Architecture & Use Case Workflow. 52
Figure 9 – Three Step Compilation Process. 55
Figure 10 – Code-autogeneration Flow Diagram. 61
Figure 11 – L2FWD Use Case Pipeline. 78
Figure 12 – L2FWD Performance Evaluation (1 core, 100 Table entries) on Testbed A. 79
Figure 13 – L3FWDv(4/6) Use Case Pipeline. 80
Figure 14 – L3FWD (IPv4 & IPv6) Performance Evaluation (1 core, 100 Table entries) on Testbed A. 81
Figure 15 – NAT Use Case Pipeline. 82
Figure 16 – NAT (UL & DL) Performance Evaluation (1 core, 100 Table entries) on Testbed A. 83
Figure 17 – Data Center Gateway (DCG) use case scenario. 84
Figure 18 – DCG pipeline featuring the UL and DL table details. 85
Figure 19 – DCG (UL & DL) Performance Evaluation (1 core, 100 Table entries) on Testbed A. 86
Figure 20 – BNG use case illustrating a subscriber and an external public service. . 88
Figure 21 – Implemented BNG pipeline featuring the main UL and DL tables. . . . 89
Figure 22 – BNG (UL & DL) Performance Evaluation (1 core, 100 Table entries) on Testbed A. 90
Figure 23 – Packet Rate for all Use Cases with different CPU Cores (128 Bytes, 100 Table entries) on Testbed A. 92
Figure 24 – Packet Rate for Different Use Cases and CPU cores (128 Bytes) on Testbed A. 93
Figure 25 – Packet Rate of different Use Cases & packet sizes. (100 Entries, 4 CPU Cores) on Testbed D 94
Figure 26 – Packet rate for different Use Cases & CPU Cores. (64 Bytes, 100 Entries) on Testbed D 95

Figure 27 – Packet rate for different Use Cases, FIB sizes. (2 core, DPDK) on Testbed E 96
Figure 28 – Packet Rate for different Use Cases, burst sizes. (100 Entries, 2 CPU cores, DPDK) on Testbed E 97
Figure 29 – Forwarding performance of different Use Cases, Pkt sizes, TG. (100 Entries, 2 Cores) on Testbed E 98
Figure 30 – Understanding Boxplot for Latency Measurements 100
Figure 31 – Latency of L2FWD DPDK Example for different packet sizes & burst sizes. (2 CPU, 99% line rate) on Testbed E 101
Figure 32 – Latency of L2FWD Use Case for different packet sizes & burst size. (100 Entries, DPDK, 2 CPU, 99% line rate) on Testbed E 102
Figure 33 – Latency of L2FWD Use Case (TX part re-implemented similar to DPDK example) for different packet sizes & burst size. (100 Entries, DPDK, 2 CPU, 99% line rate) on Testbed E 103
Figure 34 – Latency of different Use Cases, packet sizes, Fib Sizes. (64 Bytes and 1580 Bytes, 100 Entries, DPDK, 10% line rate) on Testbed E 104
Figure 35 – Packet Rate comparison of different platforms and switches for selected use cases (100 FIB size) and varying CPU cores. 105
Figure 36 – Packet Rate for L2FWD (100 entries, 64 Bytes) on Testbed B. 107
Figure 37 – Packet Rate for L3FWDv4 (100 entries, 64 Bytes) on Testbed B. . . . 107
Figure 38 – Forwarding performance for different MacS (VNF) Use Cases with different CPU Cores (64B, Testbed A). 109
Figure 39 – IPv4 and IPv6 forwarding performance of different I/O drivers, FIB size, VNFs (4 CPU cores). 109
Figure 40 – Performance (Mpps) when dynamically (30s intervals) changing the sets of CPU cores allocated to packet processing for different FIB sizes on Testbed A. 111
Figure 41 – Performance (Mpps) when dynamically (30s intervals) changing the sets of CPU cores allocated to packet processing for different FIB sizes on Testbed B. 112
Figure 42 – Residual Plot for Linear Regression 127
Figure 43 – Packet Rate (Predicted Value vs Measured Value). 129
Figure 44 – Abstract Packet Processing Pipeline 132
Figure 45 – Packet Rate comparison for all Use Cases with batch optimization for different CPU Cores (64 Bytes, 100 Table entries) on Testbed A. 134
Figure 46 – BB-Gen Architecture and Integration with NFPA and MACSAD & T4P4S 141
Figure 47 – L2FWD Dependency Graphs 159
Figure 48 – L3FWDv4 Dependency Graphs 167

Figure 49 – L3FWDv6 Dependency Graphs 168
Figure 50 – NAT Dependency Graphs 174
Figure 51 – DCG Dependency Graphs 185
Figure 52 – BNG Parser Dependency Graph 197
Figure 53 – BNG Table Dependency Graph 198

List of Tables

Table 1 – Research Directions. 24
Table 2 – ODP supported platforms 36
Table 3 – Year-on-Year Evolution of OpenFlow 45
Table 4 – Scope, Approach, and Feature Comparison List of different Programmable Switch Projects 49
Table 5 – Packet Processing Functions 53
Table 6 – P4 Object List in HLIR 57
Table 7 – Backend APIs Categorical Examples 59
Table 8 – Transformation of P4 Constructs to 'C' Language 61
Table 9 – Auto-generated Code for Header Instances 63
Table 10 – Auto-generated Code for Header Field Instances 63
Table 11 – Auto-generated Table APIs for Control Plane 73
Table 12 – Testbed Summary 78
Table 13 – Packet Rate Behavior for Different FIB Sizes 96
Table 14 – Processing Time for a Single Network Packet 98
Table 15 – Latency of BNG-UL use case for different FIB sizes & packet sizes in Testbed E 104
Table 16 – P4 Use Case Complexity Details 117
Table 17 – MACSAD Switch (MacS) Dataset Sample 119
Table 18 – Coefficient Vector of Linear Regression Model 126
Table 19 – Ridge Regression Model R^2 Scores 128
Table 20 – Regularized Model with α value as 0.001 128
Table 21 – Performance Measures of Different Regression Models 129
Table 22 – Use Case Performance Results (64 Bytes, 100 Table Entries, DPDK, Testbed A) 135

Acronyms

AAA Authentication, Authorization and Accounting.

API Application Programming Interface.

ARM Advanced RISC Machine.

BNG Broadband Network Gateway.

BRAS Broadband Remote Access Server.

CISC Complex Instruction Set Computing.

CPE Customer Premise Equipment.

DCG Data Center Gateway.

DL Download.

DPDK Data Plane Development Kit.

dRMT disaggregated Reconfigurable Match-Action Table.

DSL Domain Specific Language.

EO Elementary Operation.

FIB Forwarding Information Base.

GCC GNU Compiler Collection.

GRE Generic Routing Encapsulation.

HLIR High Level Intermediate Representation.

IR Intermediate Representation.

ISP Internet Service Provider.

JSON JavaScript Object Notation.

LLVM Low Level Virtual Machine.

LPM Longest Prefix Match.

MacS MACSAD Switch.

MACSAD Multi-Architecture Compiler System for Abstract Dataplanes.

NAT Network Address Translation.

NF Network Function.

NFPA Network Function Performance Analyzer.

NFV Network Function Virtualization.

NOS Network Operating System.

NUMA Non-Uniform Memory Access.

ODP OpenDataPlane.

OF OpenFlow.

OPX OpenSwitch.

OvS OpenvSwitch.

P4 Programming Protocol-Independent Packet Processors.

PI Protocol Independence.

PISA Protocol Independent Switch Architecture.

POF Protocol-oblivious Forwarding.

PRT P4Runtime.

RISC Reduced instruction set computing.

RMT Reconfigurable Match Tables.

RSS Receive Side Scaling.

SAI Switch Abstraction Interface.

SDK Software Development Kit.

SDN Software-Defined Networking.

SONiC Software for Open Networking in the Cloud.

T4P4S Translator for P4 Switches.

UL Upload.

VM Virtual Machine.

VNF Virtual Network Function.

VTEP Virtual Tunnel End Point.

VXLAN Virtual eXtensible Local Area Network.

Contents

1 Introduction 21
1.1 Background and Motivation 21
1.2 Research Hypothesis 24
1.3 Thesis Approach & Contributions 24
1.3.1 Multi-Architecture Compiler System for Abstract Dataplanes 26
1.3.2 Multidimensional Evaluation 27
1.3.3 Use Case Complexity Analysis 28
1.3.4 Compiler Optimization 28
1.3.5 Additional Open-source Artifacts 28
1.3.6 Noted Contributions 29
1.4 Outline 31
2 Literature Review 32
2.1 Related Technologies 33
2.1.1 Packet IOs 33
2.1.1.1 FD.io (Fast data – Input/Output) 34
2.1.1.2 Netmap 34
2.1.1.3 Data Plane Development Kit (DPDK) 34
2.1.1.4 OpenDataPlane (ODP) 35
2.1.2 Packet Switch Pipeline Architectures 36
2.1.2.1 Reconfigurable Match-Action Table (RMT) 36
2.1.2.2 disaggregated Reconfigurable Match-Action Table (dRMT) 37
2.1.2.3 Protocol Independent Switch Architecture (PISA) 37
2.1.3 High Level Domain Specific Languages 38
2.1.3.1 Pyretic 38
2.1.3.2 Protocol-Oblivious Forwarding (POF) 38
2.1.3.3 Network Assembly Language (NetASM) 39
2.1.3.4 Programming Protocol-Independent Packet Processors (P4) 39
2.1.4 Control Plane API Abstractions 44
2.1.4.1 OpenFlow (OF) 44
2.1.4.2 Switch Abstraction Interface (SAI) 45
2.1.4.3 Ethernet Switch Device Driver Model (switchdev) 46
2.1.4.4 P4Runtime (PRT) 46
2.2 Related Work 47
2.3 Concluding Remarks 50
3 Multi-Architecture Compiler System for Abstract Dataplanes 51
3.1 Architecture 51

3.1.1 Auxiliary Frontend . . . 52
3.1.2 Auxiliary Backend . . . 52
3.1.3 Core Compiler . . . 53
3.1.3.1 Transpiler . . . 53
3.1.3.2 Compiler . . . 54
3.2 Compilation Process . . . 55
3.3 P4 to IR Code Generation . . . 56
3.4 Internal & Helper APIs . . . 59
3.5 Source to Source Code Transformation . . . 60
3.5.1 Transforming Language Abstractions . . . 62
3.5.2 Auto-generating Data Path Logic . . . 66
3.6 Features of Architecture . . . 69
3.6.1 Programmability . . . 69
3.6.1.1 Protocol Independent Parser . . . 70
3.6.1.2 Protocol Independent Dataplane . . . 71
3.6.2 Portability . . . 71
3.6.3 Contoller Support . . . 72
3.7 Concluding Remarks . . . 73
4 Experimental Evaluation . . . 74
4.1 Testbed Details . . . 75
4.2 Use Case Descriptions . . . 78
4.2.1 Port Forwarding (PortFWD) . . . 78
4.2.2 Layer-2 Forwarding (L2FWD) . . . 78
4.2.3 Layer-3 Forwarding (L3FWDv4/v6) . . . 80
4.2.4 Network Address Translation (NAT) . . . 82
4.2.5 Data Center Gateway (DCG) with VXLAN . . . 83
4.2.5.1 Download (DL) . . . 84
4.2.5.2 Upload (UL) . . . 85
4.2.6 Broadband Network Gateway (BNG) . . . 87
4.2.6.1 Upload (UL) . . . 88
4.2.6.2 Download (DL) . . . 88
4.3 MacS Evaluation & Analysis . . . 91
4.3.1 Packet Rate Analysis . . . 91
4.3.1.1 Impact of FIB Sizes . . . 95
4.3.1.2 Impact of Burst Sizes . . . 96
4.3.1.3 Impact of Traffic Generators . . . 97
4.3.2 Latency Analysis . . . 99
4.3.3 Performance Comparison Against Related Works . . . 105
4.4 Performance Evaluation of MacS as Network Function . . . 108

4.5 Adaptive Scalability by Dynamic CPU Core Allocation . . . 110
4.5.1 Results Analysis . . . 110
4.5.2 Discussion . . . 112
4.6 Concluding Remarks . . . 113
5 Complexity Analysis . . . 114
5.1 Use Case Complexity . . . 114
5.2 Machine Learning (Regression) Analysis . . . 117
5.2.1 Data Processing . . . 119
5.2.2 Regression Models . . . 121
5.2.3 Regression Analysis . . . 125
5.3 Concluding Remarks . . . 129
6 Optimization . . . 131
6.1 MACSAD Packet Processing Optimization . . . 131
6.2 Evaluation and Analysis . . . 134
6.3 Concluding Remarks . . . 137
7 Open Source Artifacts . . . 139
7.1 BB-Gen Tool . . . 139
7.2 OpenDataPlane (ODP) . . . 142
7.2.1 Issues and Fixes for ODP . . . 142
7.2.2 IPv6 support for LPM Lookup in ODP . . . 143
7.2.3 Contribution for odp-thunderx . . . 143
7.3 Additional Open-source Contributions . . . 143
7.4 Concluding Remarks . . . 145
8 Future Works & Conclusions . . . 146
8.1 Future Works . . . 146
8.2 Conclusions . . . 147
Bibliography . . . 149
ANNEX A Layer 2 Forwarding (L2FWD) . . . 155
A.1 L2FWD $P4_{14}$ Program . . . 155
A.2 L2FWD $P4_{16}$ Program . . . 156
A.3 Dependency Graphs for L2FWD Use Case . . . 159
ANNEX B Layer 3 Forwarding (L3FWD) . . . 160
B.1 L3FWDv4 $P4_{14}$ Program . . . 160
B.2 L3FWDv6 $P4_{14}$ Program . . . 162
B.3 L3FWDv4 $P4_{16}$ Program . . . 164
B.4 Dependency Graphs for L3FWDv4 Use Case . . . 167
B.5 Dependency Graphs for L3FWDv6 Use Case . . . 168
ANNEX C Network Address Translation (NAT) . . . 169
C.1 NAT $P4_{14}$ Program . . . 169

C.2 Dependency Graphs for NAT Use Case 174
ANNEX D Data Center Gateway (DCG) 175
D.1 DCG $P4_{14}$ Program 175
D.2 Dependency Graphs for DCG Use Case 185
ANNEX E Broadband Network Gateway (BNG) 186
E.1 BNG $P4_{16}$ Program 186
E.2 Dependency Graphs for BNG Use Case 197

1 Introduction

Internet ubiquity paints networking devices as gateways which guides the network packets across, and depicts the task as simple and mundane. In fact, according to (CLARK, 1988), the design philosophy behind the fundamental structure of the Internet was to make it simple and easy for the Internet to grow, and to allow different networks to connect together using routers. The advancements in switch & router hardware technologies tell a different story though. To bring myriads of different networks of Internet together, and to accommodate newer requirements of data centers and other private networks, switches & routers are becoming much more complex supporting multitudes of protocols.

Although switch evolution is predominantly performance driven, recent past has seen a steady increase in the number of protocols too. supported raising the complexity of switch design. This protocol driven development process is archaic and relatively slow being of 3-5 years cycle. while slow development cycle limits the fallback option for manufacturer when the new protocol is not adopted by consumers, it also keeps increasing the complexity of hardware design making the maintenance a difficult task. This inherent inflexibility in switch design, or simply data plane design, restricts manufacturers to stay involved in proprietary switch development driving platform specific developments. Research works focusing on network flexibility are sparse at best. The difficulty in understanding the flexibility of network design and data plane design can be attributed to its multiple definitions focusing on different aspects of network. To strengthen our understanding of flexibility, our thesis tries conceptualizing innovative ideas of networking to bring flexibility to switches describable in terms of many different measures, such as *programmability, portability, performance*, and *scalability* (3PS).

1.1 Background and Motivation

Networking industry has been virtuous to borrow different technologies from other fields of research. A cursory throwback at the history of network evolution reveals that two of the promising solutions for networking hardwares namely *Disaggregation* and *Virtualization* are borrowed from other industries to bring flexibility to networking hardware, network functions and the network itself. Disaggregation was the key to 'open networking' and allowed the industry to move towards standard products with an open design to drive the networking industry. It gave birth to today's bare metal switches followed by numerous Network Operating System (NOS). By opening the hardware design, Switch costs plummeted and put hardware in the hands of the researchers and academicians. Similarly,

following the footsteps of server and storage virtualization, network professionals started rethinking networking devices as virtualized devices instead as standalone devices helping data center advancements. Then the advent of cloud pushed networking industry to adopt virtualization as a first citizen and put central control as a new requirement bringing centrally managed networks. Although these developments did not directly translate to bringing programmability to devices for which SDN strives for, they indeed set the foundation for our current research discussions and for the future networks.

In general, any packet switch[1] architecture comprises of two main components consisting of data plane and control plane where data plane is responsible for packet forwarding, while control plane serves data plane to define the datapath and its rules. Software-Defined Networking (SDN) (KREUTZ *et al.*, 2015) paradigm, a new school of thought, advocates separation of the control plane from the data plane in the switch. It allows programming the network control plane by managing the routing protocols on a logically centralized server instead of at the switch. To truely adopt SDN paradigm, a equally similar level of flexibility is necessary from data plane too. However, only limited programmability was also brought to data plane by the programmable chips. Programmable chips bring flexibility, specifically programmability, to switch in terms of loosely defined tables characterized by its size, lookup algorithm, and so forth. To be exact, OpenFlow protocol (MCKEOWN *et al.*, 2008) became fundamental in guiding the industry to popularize and adopt match + action abstractions to configure switch tables, and can be considered as a *de facto* standard for defining programmable data planes facilitated by programmable chip designs. As a result of this movement, some equipment vendors successfully released products in limited numbers supporting SDN and OpenFlow. But none the less a data plane design independent of supported protocol and underlying target remained far from reach.

This brief history leads us to our current research work and discussion. In a conventional network, each comprising network device maintain a set of network applications running routing algorithms to generate the network rules for the network traffic to follow. The control plane of these devices are responsible to propagate these rules to the data planes which simply forward packets accordingly. The co-existence of both control plane and data plane has limited the development to the opening and standardization of the APIs between them. *Disaggregation* bridles the inflexibility to a small extent in designing the network devices and brings support for different NOSs ushering *Portability*. Furthermore, following SDN principle, control plane and data plane i.e., software and hardware entity of switches can be separated in every network device in a network by centralizing all the control planes in a separate unified logical entity leaving the data planes to operate individually bringing programmability to the network architecture over its control plane.

[1] Switch is used to refer both switch and router in this thesis as modern literature considers switches as L2-L7 switches.

And we envision the proliferation of *Programmability* into the less explored entity, the data plane too; realization of a fully flexible network device. With this we bring focus to the flexibility requirements in terms of Portability and Programmability of switches.

Data plane programmability related research mainly focuses on three different levels: data plane architecture, domain specific language and low-level SDK to define a data plane. Current advancements in programmable data plane architectures, Reconfigurable Match Tables (RMT) (BOSSHART *et al.*, 2013), disaggregated Reconfigurable Match-Action Table (dRMT) (CHOLE *et al.*, 2017), or Protocol Independent Switch Architecture (PISA) (MCKEOWN, 2015) to name a few, promise to offer flexibility in data plane allowing post-fabrication reconfiguration by manufacturers and consumers. In a simple manner, a switch data plane extracts network packet header informations and matches the extracted information against the flow table rules followed by performing associated actions before forwarding the network packet out. These two packet processing activities, also known as header parsing & table lookup, are identified as design abstractions in data plane architectures: parser and match + action table abstractions. Domain Specific Languages (DSLs) such as Protocol-oblivious Forwarding (POF) (SONG, 2013) and Programming Protocol-Independent Packet Processors (P4) (P. Bosshart et al, 2014) understand these design abstractions and offers intuitive language constructs to implement these design abstractions. Meanwhile, OpenDataPlane (ODP) (OPENDATAPLANE, 2018), a powerful low-level SDK, provides a compiler system necessary to define a target switch using these design abstractions. In a nutshell, we feel the necessity of a unified multi-architecture compiler system comprising of compatible set of data plane architecture, DSL, low-level SDK, and a target compiler crucial for flexible programmable data plane.

Introduction of programmability into data plane has its own side effects and brings new challenges. Data plane programmability gives away with the fixed protocol set, and allows consumers and operators to define custom protocols transforming the data plane from fixed-function *Protocol Dependent* into *Protocol Independent*. Flexibility in data plane brings challenges to control plane which until recently have been benefited by standardized Application Programming Interfaces (APIs) towards data plane. With custom protocols in place, the control plane needs to adapt to the pipeline definition every time the consumer redefines it. This gives rise to a new school of thought which advocates a alternate way to define a pipeline which can be input to both data plane and control plane, and helps in defining the messages and APIs among them. P4Runtime (PRT) (P4API, 2018), Openconfig, etc are some of the solutions currently available exploring this ideology. This allows a control plane to control any forwarding plane regardless of what protocols and features the underlying data plane supports.

1.2 Research Hypothesis

In the previous section, we explored data plane flexibility and the three aspects of data plane development. We have also identified different aspects of flexibility such as *programmability, portability, performance*, and *scalability* (3PS). The advancements in data plane architectures, availability of supported DSL to implement data plane design abstractions, and a powerful low-level SDK over a compiler system are the dictating factors to achieve a flexible data plane. By defining a data plane using the 'parser' & 'match+action' design abstractions from data plane architectures, it is possible to have a protocol independent data plane. We identify protocol independent nature as *Programmability* aspect of Flexibility in a data plane. Following up, by adopting a multi-architecture low-level SDK we can bring the same data plane to different platforms while focusing on portability aspect of flexibility of the data plane. While Programmability & Portability are related to the design of the data plane, Performance & Scalability aspects of flexibility are more apt for the evaluation of the data plane. Simply put, our research encompasses both design and evaluation of flexible data plane. To define, our research hypothesis to aim for would be, ***An open-source multi-architecture compiler system towards data plane flexibility satisfying the ever so important contending features 3PS; programmability, portability, performance, and scalability; in our advancement to the future networks.***

1.3 Thesis Approach & Contributions

The breadth of this research proposal spans along data plane flexibility while ignoring the interaction towards the control plane. Being said that, the primary focus is towards the practical aspects of building our proposed compiler system, MACSAD, with open-source or free components (whenever possible) available addressing data plane features and a thorough evaluation of MACSAD addressing different aspects of flexibility (i.e., 3PS). The contributions of this thesis includes different facets of MACSAD development and evaluation: A muti-architecture compiler system, MACSAD, to achieve flexible data plane; Evaluation of MACSAD inline to different aspects of flexibility; Complexity analysis of different use case pipelines & performance prediction using machine learning algorithms; Additional compiler optimization for MACSAD performance improvement; Multiple open-source artifacts developed along with MACSAD development. All the contributions are briefly explained and summarized in Table 1.

Table 1 – Research Directions.

Continuation of Table 1

multi-architecture compiler system

Our Solution: We explore our proposed **MACSAD** compiler system which can define a protocol-independent and target-independent pipeline by bringing P4 and ODP together.

Challenges: Lack of SOA. Slim research community.
Missing supporting tools such as benchmarking.

Explored In: chapter 3.

Related Publication: (PATRA *et al.*, 2016)

multidimensional evaluation of MACSAD

Our Solution: MACSAD is evaluated for programmability, portability, performance, and scalability (3PS). We also evaluate MACSAD as VNF towards functional scalability. Finally we evaluate resource scalability with adaptive dynamic CPU allocation method.

Challenges: Creating similar test bed for different target architectures.
Choosing a common tool across architectures for benchmarking.

Explored In: chapter 4.

Related Publications: (PATRA *et al.*, 2017)(PATRA *et al.*, 2018)

use case complexity analysis

Our Solution: We compiled a list of complexity factors for different use cases. We apply Regression based Machine Learning algorithms to create a model and predict performance of MACSAD.

Challenges: Lack of SOA related to complexity analysis. Difficulty in collecting data for Machine Learning algorithm.

Explored In: chapter 5.

compiler optimization

Our Solution: We improved MACSAD code auto-generation phase maximizing CPU-memory parallelism.

Challenges: Identifying common approach to code auto-generation for different pipelines where the underlying compiler can optimize the generated code efficiently.

Explored In: chapter 6.

Continuation of Table 1

additional open-source artifacts

Our Solution: We created BB-Gen packet generator tool, and used it for all our evaluation experiments. We have multiple contributions as bug fixes and feature additions to P4 and ODP code bases.

Challenges: Necessity to have modular and user-friendly code base for BB-Gen. Adding P4 support for BB-Gen. Developing competence over the complex P4 and ODP code base to contribute.

Explored In: chapter 7.

Related Publications: (RODRIGUEZ *et al.*, 2018) (CESEN *et al.*, 2018a)

End of Table 1

1.3.1 Multi-Architecture Compiler System for Abstract Dataplanes

Our thesis proposal Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD) blends the Top-Down approach of P4 towards protocol-independence and Bottom-Up approach of ODP towards target-independence. It is a cross-platform compiler system which incorporates low-level but target-independent (HW & SW) APIs from ODP to offer data plane over various network platforms including CPUs (flexible data path, lower performance) based on Complex Instruction Set Computing (CISC) like x86, Reduced instruction set computing (RISC) like Advanced RISC Machine (ARM) (highly multi-core) etc,. In addition, MACSAD brings support to different packet I/Os too. In the big picture,dd MACSAD incorporates support for a number of features as shown in Figure 1 and provides us with numerous opportunities, albeit challenging, for carrying out different research activities.

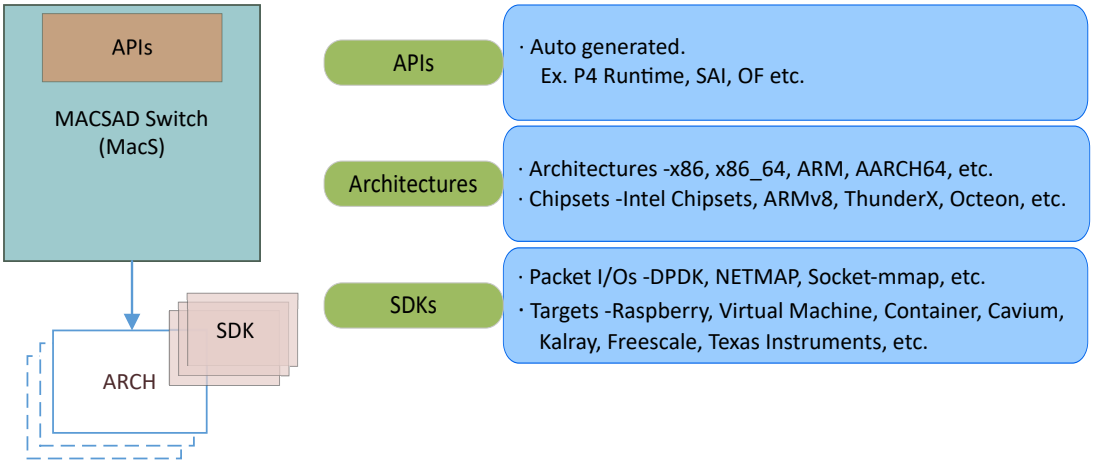


Figure 1 – Supported Features envisioned by MACSAD

1.3.2 Multidimensional Evaluation

We evaluate MACSAD with respect to Programmability, Performance, Portability, and Scalability (3PS) factors as explained here.

Programmability of MACSAD is affirmed by various use case pipelines presented in (PATRA *et al.*, 2018), (MEJIA *et al.*, 2018), (CESEN *et al.*, 2018b). In our effort towards protocol-independence, we bring diverse use cases with increasing complexity in terms of number of table lookups and table actions, and present a detailed evaluation of the use cases. The use cases supported on MACSAD include Layer 2 Forwarding (L2FWD), Layer 3 Forwarding (L3FWDv4/L3FWDv6), Network Address Translation (NAT), Data Center Gateway (DCG), and Broadband Network Gateway (BNG), to name them all.

Portability of MACSAD is explored by bringing the aforementioned use cases to different platforms like x86, ARMv6, and ARMv8 spanning Intel Servers, Raspberry Pi2, and Cavium switches. Apart from showing the feasibility of running MACSAD based data planes, we also evaluate the performance of data planes on the supported target platforms.

Performance & Scalability of MACSAD are evaluated and measured in terms of packet rate and latency of different use cases. We explore the performance results of on different platforms with a varied number of CPUs exploring the scalability aspect. We also put MACSAD against two related works (such as the P4 based switch *T4P4S* (LAKI *et al.*, 2016) and the DPDK-capable production quality open source software switch Open-vSwitch (OvS) (PFAFF *et al.*, 2015)) from Table 4 and evaluated their performance over different use cases.

In addition to the 3PS, we also explore two other aspects flexibility during evaluation of MACSAD. We evaluate the *resource scalability* by analyzing a novel technique providing dynamic CPU scaling through run-time (de)allocation of CPU cores in MACSAD data plane. Scaling up/down can be adaptive based on system load, on traffic workload, or other factors (e.g., energy consumption). Such behavior is instrumental in a multi-tenant environment, where de-allocated CPU cores could be used for other tasks.

We stretched our evaluation activities to include functional scalability too. Functional scaling is achieved by deploying multiple instances of the network function, MACSAD data plane in our case, to achieve higher performance. This behavior is more prominent in a Network Function Virtualization (NFV) environment where scaling is achieved by instantiating a network function in multiple. We present MACSAD as a Virtual Network Function (VNF) and carry out the performance evaluation for the same to provide a glimpse into how MACSAD will behave in a NFV environment.

1.3.3 Use Case Complexity Analysis

For evaluation of a switch or a Network Function (NF), it is necessary to build a methodology to identify the key components and factors (a.k.a. Complexity Factors) influencing the performance. This gives us an insight into the complexity of the use cases and opens more opportunity to bring complexity into consideration while discussing performance, portability or scalability. With sufficient information, it might be possible to come up with techniques to bring performance improvements too to the use cases. An earlier work (SAPIO *et al.*, 2015) tried to measure the performance of NFs by identifying recurring execution patterns Elementary Operation (EO) and mapping them to the hardware. This work solely focuses on measuring performance in terms of packet rate. This gives us a glimpse into the Complexity Factors responsible for a NFs. We present our take on complexity analysis of P4 based pipelines and extend it to MACSAD. Based on (DANG *et al.*, 2017), we have identified our *Complexity Factors* from the P4 programming language constructs. We explored all the use cases supported by MACSAD, and present the complexity details for all the P4 programs in Table 4. Then, we bring machine learning algorithms to analyze the complexity of the use cases using the *Complexity Factors* as features. We use Regression methodologies to learn the relationship between the use case complexities and their performance, and train different machine learning models to predict the performance of a MACSAD use case from its P4 program. This will allow to predict performance a data plane defined by a P4 program even before compiling it over to the target platform.

1.3.4 Compiler Optimization

MACSAD implements a number of optimization techniques across its different modules. However, our work towards exploiting the memory-level parallelism between CPU and main memory (BHARDWAJ *et al.*, 2017) is important due to its clear impact on performance by targeting the memory-bound steps of packet processing, i.e., the table lookup step which consists of table key creation and the actual lookup step. Taking into account that the steps involved in different types of table lookups are most of the times similar, we implement batched table key creation and table lookup to exploit the memory level parallelism while hiding the CPU-memory latency. As explored in (WANG *et al.*, 2018), more than 70% of packet processing time is spent on table lookup in the datapath, and therefore this task focuses explicitly on table lookup.

1.3.5 Additional Open-source Artifacts

Our work with MACSAD has pushed us to work on different ideas and projects giving rise to multiple contributions to the research community. We faced many difficulties in procuring test traffic data to evaluate MACSAD use cases. This inspired us to

come up with our own tool BB-Gen (RODRIGUEZ *et al.*, 2018) to overcome the hurdles towards agile data plane performance evaluation by its simplicity and effectiveness to generate network traffic and P4 table entries for different P4 use cases with augmenting complexity. Our other contributions are more focused on P4 and ODP source code and their features. We have contributed with new features such as IPv6 based LPM lookup method, and an extension to dependent graph generation module to the ODP and P4 repositories respectively. Apart from this, we have offered our help in providing a testbed for bug reproduction, for validating the patches for bug fixes, and also directly contributing patches to fix issues in ODP and P4 code repositories. In addition to these, we have provided all our research artifacts as open source for the research community to take advantage of.

1.3.6 Noted Contributions

We present here all the contributions in terms of scientific publications accomplishing the breadth of this thesis from different fronts. All the collaborative efforts are described and referenced to the corresponding scientific article indicating the co-authors and their contributions. The inclusive list of publications is shown below.

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

(B) Towards Realization of High Performance Programmable Datapaths using Domain Specific Language, *Décimo Primeiro Encontro dos Alunos e Docentes do Departamento de Engenharia de Computação e Automação Industrial (XI EADCA),Campinas, Brazil, 2018,* P. Gyanesh Patra, C. E. Rothenberg.

(C) BB-Gen: A Packet Crafter for P4 Target Evaluation, *ACM SIGCOMM'18 Demo and Poster Session, 2018,* F. R. Cesen, P. Gyanesh Patra, L. Csikor, C. Rothenberg, P. Vörös, S. Laki, and G. Pongrácz.

(D) Design, Implementation and Evaluation of IPv4/IPv6 Longest Prefix Match support in P4 Dataplanes, *Csbc 2018 – 17º wperformance, 2018,*F. R. Cesen, P. Gyanesh Patra, C. E. Rothenberg, and G. Pongrácz.

(E) MACSAD: An Exemplar Realization of Multi-Architecture P4 Pipelines, *5th P4 Workshop, June 2018,* P. Gyanesh Patra, F. Rodriguez, J. Mejia, D. Feferman, C. Rothenberg and G. Pongrácz.

(F) BB-Gen: A Packet Crafter for Performance Evaluation of P4 Data Planes, *5th P4 Workshop, June 2018,* Fabricio Rodriguez Cesen, P. Gyanesh Patra, Christian E. Rothenberg, Gergely Pongrácz.

(G) BB-Gen: A Packet Crafter for Data Plane Evaluation, *Salão de Ferramentas, 36th Brazilian Symposium on Computer Networks and Distributed Systems (SBRC 2018), May 2018,* F. R. Cesen, P. Gyanesh Patra, and C. E. Rothenberg.

(H) MACSAD: high performance dataplane applications on the move, *18th IEEE international conference on high performance switching and routing (HPSR), Brazil, 2017,* P. Gyanesh Patra, C. E. Rothenberg, and G. Pongrácz.

(I) MACSAD: Multi-Architecture Compiler System for Abstract Dataplanes (aka Partnering P4 with ODP), *Acm sigcomm'16 demo and poster session, August 2016,* P. Gyanesh Patra, C. E. Rothenberg, and G. Pongrácz.

The complete effort around MACSAD incorporates a number of collaborative activities which comprises tasks such as writing of P4 programs, contributing to open source projects and more. We acknowledge the contributions towards the development of P4 programs of the use cases such as L3FWDv6, Data Center Gateway (DCG), and Broadband Network Gateway (BNG) by the co-authors from item (A) (PATRA *et al.*, 2018). In addition to the P4 programs, efforts contributing towards MACSAD source code in line to the DCG and BNG use cases are also acknowledged here. Likewise, item (D) shows the contributions to MACSAD implementing L3FWDv6 use case by the co-author. The contributions from the collaborators helping in carrying out the experiments, and collecting the results for burst size analysis and latency evaluation of MACSAD are detailed in chapter 4. Furthermore, a big credit to the co-author of (CESEN *et al.*, 2018b) and (RODRIGUEZ *et al.*, 2018) for the contributions towards additional artifacts of MACSAD (subsection 1.3.5) identified as IPv6 based LPM support for ODP and BB-Gen. Continuing on acknowledgments, we want to mention the Translator for P4 Switches (T4P4S) project team as the MACSAD development was bootstrapped from a part of the seed code that we shared with the initial phase of T4P4S project and became a part of Transpiler sub-module of MACSAD mentioned in subsubsection 3.1.3.1. Finally, We also acknowledge Ericsson Research Brazil for the financial support, and Ericsson Research Hungary for their support during the development of this thesis proposal.

We are delighted to receive contributions touching upon different aspects of MACSAD, and at the same time also immensely satisfied to be able to contribute to other's works as part of this thesis proposal and related tasks.

1.4 Outline

We begin with the state of the art and literature review of related technologies and related works presented in chapter 2. Following on, this thesis explores different problem classes while explaining the design and development of our proposed MACSAD (PATRA *et al.*, 2016; PATRA *et al.*, 2017; PATRA *et al.*, 2018) project in chapter 3. We present our evaluation of MACSAD around four characteristics of flexibility, namely, *performance, portability, programmability* and *scalability* (3PS) in chapter 4. We explore data plane complexity of different use cases followed by complexity analysis using machine learning algorithms in chapter 5. Then we bring our findings on compiler optimization activities in chapter 6. In chapter 7, we explain all our additional contributions that came out from our research activities which include an open source tool, addition of features to open source code repositories, bug fixes and many more. Finally, we present our conclusion with remarks for future goals and activities in chapter 8.

2 Literature Review

Programmable switches go way back to the beginning when the first ever switch, also known as Interface Message Processors (IMPs), was implemented in software with an initial data rates of 56kbit/s. Broad adoption of internet and growing demand of the World Wide Web (WWW) pushed the bandwidth requirement which was not feasible with the software switches anymore. In 1998 Juniper brought to market the wire-speed ASIC based router *M40* with ten times the throughput of comparable contemporary Cisco products like *CISCO 12000*. After this, we see a flurry of ASIC based switches came out providing higher throughput year on year. Switches adhere to a vertically integrated two layers design with control plane and data plane, and referred as fixed-function devices. Although manufacturers kept adding more programmability to the hardware, it was always to support newer features whereas switches remain fixed-function for the consumers and operators.

OpenFlow (OF) (MCKEOWN *et al.*, 2008) came as a new effort from the research community to bring flexibility to the data plane. It permits flow entry updates of the switch lookup tables at runtime using standardized interfaces. Although this brings configurability to the data plane, OF still is restricted with its reliance on the fixed header structures of the supported standardized protocols. But, SDN (KREUTZ *et al.*, 2015) sought to break the vertical integration and advocates decoupling of control and data plane in a switch. It also takes the control plane to a centralized server bringing programmability to the control plan. A SDN control plane can control, configure and manage a whole network as it can have a network-wide view instead of a standalone device local view. Without vertical integration in a switch, and with sufficient programming capability of the hardware made the research community to rethink the internet in terms as it was created, i.e., over software. However, we extend the thought and present the idea as protocol-independent and target-independent data plane instead of just software based. We focus our discussion on the switch data plane aligning the discussions to our thesis.

Data plane programmability is reimagined after the match+action table abstraction popularized by OF. Different DSLs (such as POF (SONG, 2013), P4 (P. Bosshart et al, 2014), Frenetic (FOSTER *et al.*, 2011), etc,.) tried to present the data plane in terms of design abstractions based on match action abstractions, and are explained later in this chapter. RMT (BOSSHART *et al.*, 2013) and dRMT (CHOLE *et al.*, 2017) are some of the packet switch architectures for designing the data plane which explored these abstractions in a more detailed manner as explained later. These DSL based design abstractions and packet switch architectures bring newer way to define protocol-interdependent and target-independent data plane which does not depend on any fixed header structures or

protocols. This provides researchers an opportunity to run experimental protocols in their switches and internal networks.

Without the fixed protocols the controller or control plane are not aware of the data plane constructs to configure or manage the data plane. As a result project like PRT (P4API, 2018) came out of incubation which is a protocol-independent API and can be auto-generated from data plane definitions written in P4 DSL. PRT envisions to be able to control any data plane and remain auto updated when data plane features changes. We will present a brief description of PRT in this chapter for the sake of completeness of the discussion though PRT is out of the purview of this thesis.

Apart from these a number of different tools and projects are also discussed which are essential in shaping our research proposal. We present the state of the art under two different sections such as "related technologies" and "related works". *Related Technologies* section includes the tools and projects which has been part of our proposal project and been an influencer. Similarly, *Related Works* section presents the many projects form the community which are more closer to our research proposal among the state of the art and had direct or indirect impact towards finalizing this thesis.

In a nutshell, our research proposal integrates PISA (MCKEOWN, 2015) (subsubsection 2.1.2.3) design abstraction, P4 DSL (subsubsection 2.1.3.4), and ODP (OPENDATAPLANE, 2018) (subsubsection 2.1.1.4) packet IO framework under the MACSAD umbrella system to create programmable software switch for multiple target architectures.

2.1 Related Technologies

Given the reborn interest in network programmability through SDN and the growing interest in data plane abstraction activities (e.g., P4, SAI, ODP), we focused ourselves on studying the emergent ecosystem of data plane abstraction technologies. Feasibility studies along with performance and portability comparison among various data plane abstraction technologies and HW targets will provide the required understanding about where the current solutions stand at and where we are heading to. Results from these studies contribute to the roadmap of our proposal by contributing knowledge about different technologies and in general contributing towards making more informed technological decisions.

2.1.1 Packet IOs

Until recently the software-based packet forwarding was limited by the capability of Linux kernel based packet forwarding infrastructure. Soon it became clear that to achieve better packet rate it is necessary to take the feature out of the kernel and provide more flexibility to the user. Features such as userspace memory-mapping to the packet buffers

of NICs and introduction of hugepage memory system were a boost to this development giving rise to fast packet processing solution like DPDK (DPDK. . . , 2010). We witness a number of new advancements in relation to different packet IO frameworks leading the race headed for high forwarding throughput in programmable software switches.

2.1.1.1 FD.io (Fast data – Input/Output)

FD.io is the recent entrant and brings several open source projects and libraries to support flexible and programmable data plane application on a generic hardware platform. It can deliver high-throughput and low-latency services over different architectures and deployment environments. The key component of FD.io would be the Vector Packet Processing (VPP) library donated by Cisco which is extremely modular and allows adding new capabilities as additional graph nodes with zero modification to the underlying code base. It was adopted under The Linux Foundation Networking Fund ("LFN") in January 2018 demonstrating the confidence of the Linux community for this project and providing long term support for the code base.

2.1.1.2 Netmap

Netmap (RIZZO, 2012) is also a fast packet processing I/O framework implemented as a kernel module. It allows the data applications to work using *netmap* seamlessly driver when available without requiring any changes to the applications. It offers zero-copy, batched IO and other features while limited by the absence of any APIs supporting inherent hardware acceleration. It achieves high performance by implementing memory mapping to the packet buffers of NICs. Netmap drivers exist completely in the kernel, and the system does not rely on IOMMU or other special mechanisms. It is a clean solution without disrupting the Linux Kernel-based packet IO framework and is integrated by the BSD kernel. It can be a go-to solution if upstream support of Netmap for relevant NIC drivers and kernel developers can be possible.

2.1.1.3 Data Plane Development Kit (DPDK)

Data Plane Development Kit (DPDK) (DPDK. . . , 2010) is the most commonly used and widely adopted user space fast packet processing IO driver collection used for defining data plane and fast packet processing on a wide variety of CPU architectures. Started by Intel in 2010, it is currently an open-source project under the Linux Foundation. It offers a multi-core framework for users to build vendor-neutral software and data plane applications. DPDK is heavily optimized for Intel® architectures. It uses hugepages to reduce TLB flushes and achieve higher packet throughput performance. It also implements features like zero-copy, batched I/O and Non-Uniform Memory Access (NUMA) support.

2.1.1.4 OpenDataPlane (ODP)

OpenDataPlane (ODP) (OPENDATAPLANE, 2018), a new entrant, has emerged to provide an abstract APIs specification to support Linux based network applications. ODP establishes a set of higher level common APIs spanning equivalent features across multiple targets mentioned in Table 2 making data plane applications portable. ODP can be compared to OpenGL as being the common standard for programming networking devices instead of video graphics. ODP establishes itself as a higher abstraction than DPDK and Netmap, and provide support for them as underlying fast packet IO technologies. It extends the highly-optimized vendor-specific Software Development Kits (SDKs) while abstracting the hardware acceleration features (e.g., Crypto) of the underlying hardware. OpenDataPlane project is created to offer an open-source, and cross-platform set of APIs for any networking data plane. Two important components of ODP are ODP API specification and ODP API implementation.

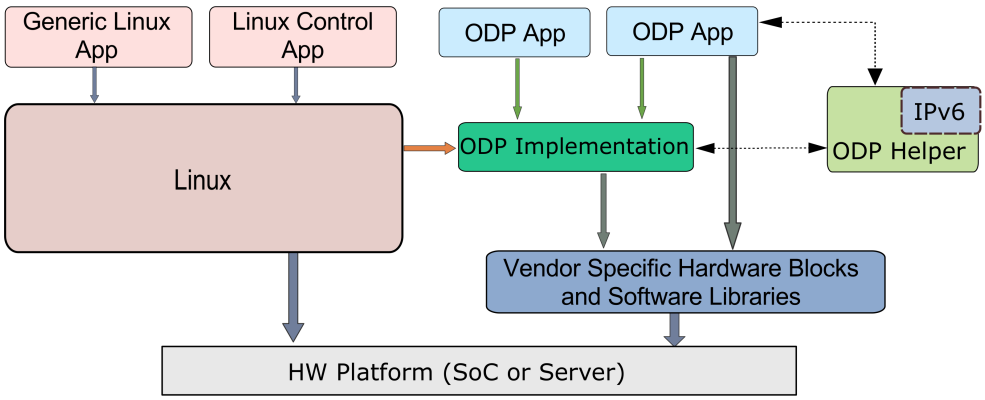


Figure 2 – Where ODP is situated?

ODP API specification describes a functional model for data plane applications. It covers the common features across multiple targets and also common programming requirements. Basic data plane application programming requirements such as packet receive and send (also known as Packet IO) are defined under the specification without specifying their implementation. It goes beyond this by describing the ODP APIs using abstract data types leaving their definition to the ODP implementers. For example, ODP packets are referenced by abstract type odp_packet_t whereas the actual implementation of it is the responsibility of ODP implementers.

Under current practice, the ODP implementations available for different platforms are implementations of the ODP API specification tailored for each platform. This design practice allows hardware offloading to be implemented for some APIs in a specific platform which might not be possible in another platform. From the application point of view, the underlying functional behavior is independent of the platform level implementation details of the ODP APIs. This is very important as ODP thrives on the balance of ODP Implementations to be open sourced vs. left up to the semiconductor vendor. The vendors

decide whether to opt for open sourced or proprietary implementation of the ODP APIs. Developers can write data plane applications without being an expert of the underlying platform only by confirming to the ODP API specification.

Figure 2 shows the scope of ODP in a switch platform complementing the vendor specific SDKs by providing common set of APIs transforming ODP portable across supported platforms mentioned in Table 2. The blue rectangle in Figure 2 shows how the data plane applications and ODP APIs co-relate to the vendor specific hardware blocks and libraries. We use ODP as part of our compiler system to bring portability for the data plane applications.

Table 2 – ODP supported platforms

Company	**Supported Platforms**	**Architecture**
Cavium Networks	Cavium Octeon^{TM} SoCs	MIPS64
	Cavium ThunderX^{TM} SoC	ARMv8
Kalray	MPPA platform	MPPA
Hisilicon	Hisilicon Platform	ARMv8
Freescale	QorIQ SoCs	Power & ARMv8
Texas Instruments	TI Keystone II SoCs	ARM Cortex-A-15
Marvell	Marvell ARMADA 8K SoCs	ARM Cortex-A72
Linaro	Uses DPDK as pacet I/O acceleration layer.	Intel x86
	Not a performance target. Reference for any Linux kernel. Software-based with Netmap & DPDK support.	Any

2.1.2 Packet Switch Pipeline Architectures

Traditionally the high-speed switch pipeline is composed of multiple fixed stages of match-action where each stage is responsible for a specific packet processing operation like extract MAC_{dest} and perform a lookup for this address. The supported protocols are the result of these stages presented in a pipeline format which little to no possibility to make any changes by the consumer. However, with programmable switch development in place, it is a requirement for the switches to offer an option to program the match-action stages using different DSL. Programmable switches have offered a lower performance compared to fixed-function switches. We present here various switch pipeline architectures evolved to bring programmability without sacrificing performance.

2.1.2.1 Reconfigurable Match-Action Table (RMT)

For SDN switches OF specification brings flexibility to define the switch pipeline by using multi-table matching. At the same time it can not configure the width, depth or number of table of the underlying platform, and is also limited by the header structures

supported (e.g., Ethernet, IP, UDP) for parsing and matching and actions supported (e.g., forwarding, dropping, decrementing TTLs, pushing VLAN header) to process packets. The inability to extend a fixed-function switch in use contributes to the limitation of OF. As an answer, RMT (BOSSHART *et al.*, 2013) explores the multi-table matching architecture to bring programmability to switch pipeline. RMT defines abstracts for header parsing and also represent the table actions in an abstract way which allow supporting custom protocol headers and custom actions on any header fields as necessary. RMT uses match-action stages to define a pipeline where each stage consists of 3 components such as (1) match component to extract header and create match keys, (2) Table component with flow entries for lookup, (3) Action component to process and modify packet fields and headers. RMT architecture brings multiple stages in a sequence to create a packet process pipeline.

2.1.2.2 disaggregated Reconfigurable Match-Action Table (dRMT)

dRMT (CHOLE *et al.*, 2017) addresses the limitations and improves over RMT in defining programmable data plane pipeline. RMT pipeline stages has local memory to define tables which may not be sufficient for a large table which can end up spreading over multiple stages resulting in poor resource utilization. In another note, RMT defines a fixed pipeline where a packet follows the stages sequentially. This may result in under-utilization of resources when in a stage only default action is defined without any preceding match, or when a packet has to traverse through all the stages in cases of recirculation independent of the actual number of stages necessary. dRMT creates a pool of memory and a cluster of processors to dynamically adapt them according to the pipeline. This allows defining a pipeline without any fixed order of the stages with flexible memory and processor allocation for each stage.

2.1.2.3 Protocol Independent Switch Architecture (PISA)

Bottom-up from the data plane perspective, Protocol Independent Switch Architecture (PISA) (MCKEOWN, 2015) is becoming the obvious approach for programmable hardware. PISA architecture paradigm resides at a higher abstract level than RMT and dRMT, and brings programmability to users by going contrary to the traditional wisdom that programmability always comes with a cost in terms of performance. Unlike RMT & dRMT, PISA defines higher level generic abstractions for packet pipeline as shown in Figure 3. It defines a pipeline as a chained set of match+action table abstraction preceded by a programmable parser where each stage can accommodate a single table or multiple tables which may or may not include a partial table spreading from a previous stage. PISA can deliver rich flexibility without compromising performance for comparable chip area and energy consumption. PISA identified various primitive instruction sets for processing packets based on which data plane applications can be written using a high-level DSL such as P4 to define and configure the underlying packet pipeline. To summarize,

PISA advocates that a programmable data plane can be defined by configuring the underlying tables, and, in turn, supporting (re-)configuration of data plane at a far later stage unlike during the fabrication phase as is the case of traditional networking ASICs while keeping the pipeline simple and performance at par.

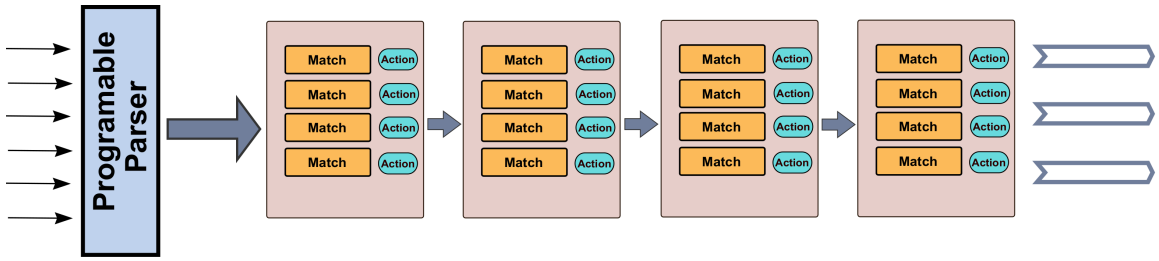


Figure 3 – PISA Architecture. Based on:(MCKEOWN, 2015)

2.1.3 High Level Domain Specific Languages

Continuing with our discussion over programmable data plane, we bring focus to the importance of high-level DSL to define a data plane. Currently, switches require a priori knowledge of the protocol header format and application semantics for the control plane and data plane to work seamlessly. Otherwise known as protocol dependent switch architecture, this brings out the fundamental flaw hindering the support for SDN. DSL can introduce design abstractions to define a data plane supporting custom protocols and header fields, and does not mandate any priori knowledge of any protocol. We explore here a number of DSLs developed towards achieving protocol independent data plane.

2.1.3.1 Pyretic

Pyretic (MONSANTO *et al.*, 2013) introduces abstractions to build SDN applications from multiple, independent modules represented as network policies to manage network traffic. The network policies pass through parallel or sequential composition before being executed on an abstract network topology reproducing the constraints applicable to the modules. It also introduces an abstract packet model which introduces virtual fields on the packets supporting packet metadata. Pyretic can even be used to design large, sophisticated controller applications comprising of smaller modules. Briefly, Pyretic can be seen as a language, and a system able to compose network policies representing SDN applications in different ways and execute them on an abstract network topology to evaluate.

2.1.3.2 Protocol-Oblivious Forwarding (POF)

Protocol-oblivious Forwarding (POF) (SONG, 2013) focuses on removal of protocol-specific configurations from the forwarding devices to achieve programmable data plane.

It defines a concise set of protocol independent and platform independent instructions known as Flow Instruction Set (FIS) which can be used to define any network service turning the forwarding device into protocol oblivious. POF assembles the search keys from the packet header, perform the table lookups, and then perform the associated table action function and related instructions. The packet processing is done under the guidance of the controller through a sequence of generic key assembly and table lookup instructions. It also decouples the match and action function of each flow and allows reuse of action functions across multiple flows tables. We consider POF to be one of the first solutions with a vision towards a fully programmable data plane to achieve true flexible SDN implementation.

2.1.3.3 Network Assembly Language (NetASM)

The concept of an Intermediate Representation (IR) is not new in the domain of compiler technology. While understanding the different pipeline architecture and DSLs, we realize that there is a place for an IR when we bring DSL to the target platform. To fill the gaps between higher-level programming languages (e.g., P4, POF, etc.) and the underlying hardware targets, NetASM (Network Assembly Language) (M. Shahbaz et al, 2015) has been proposed for programmable data planes. NetASM is a low-level intermediate programming language providing a 1-to-1 correspondence with the underlying platform based on well-defined constructs to define various low-level packet operations. NetASM enables some optimization methods to improve the performance and resource utilization of data plane applications. Currently, a prototype of NetASM is available by the developer which provides limited support for $P4_{14}$.

2.1.3.4 Programming Protocol-Independent Packet Processors (P4)

Programming Protocol-Independent Packet Processors (P4) (P. Bosshart et al, 2014; BUDIU; DODD, 2017) is a high-level declarative language which can use high-level network abstractions to express packet processing pipeline for any data plane. P4 development is motivated by three primary goals as (1) Protocol Independence, (2) Target Independence and (3) (re-)configurability of a target. P4 abstractions includes header, table, action etc., to define a data plane pipeline confirming to the abstract model at Figure 4. P4 supported abstract model consists of a parser & match+action tables sandwiched between ingress and egress. When a packet arrives, the headers are parsed and then passed through the match+action table resources carrying out lookup over header fields and applying actions to the packet headers upon match in the table followed by the deparser to serialize metadata into the packet, and finally send the packet out at egress. P4 achieves 'Target Independence' as the P4 programs are built for PISA in Figure 3 confirming to the P4 abstract model. Similarly, protocol independence is achieved by the 'header' abstractions which allow to define any arbitrary network protocol header for the

programmable parser of the PISA model. P4 being an abstract programming language achieves (re-)configurability on a programmable data plane as the target and protocol independence traits help to change the packet forwarding behavior and reprogram the target device.

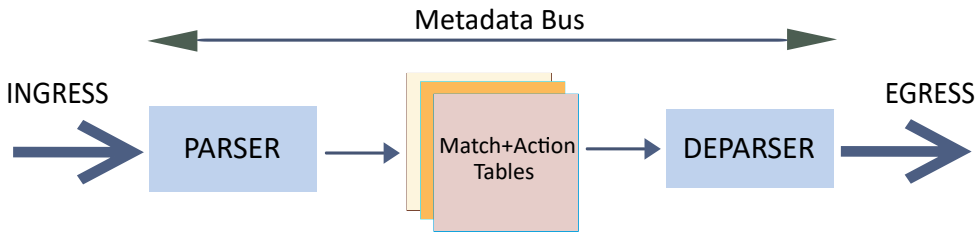


Figure 4 – P4 Abstract Forwarding Model

P4 language consortium has recently released a newer version of P4 called $P4_{16}$ while the previous version is referred to as $P4_{14}$. $P4_{16}$ brings language and architecture separation in support of a new Portable Switch Architecture (PSA) to bring $P4_{16}$ support to more diverse platforms, unlike $P4_{14}$ which supports only PISA architecture. Extern function support is added to $P4_{16}$ instead of a fixed set of primitive actions as in $P4_{14}$. This allows programmers to define different action functions corresponding to the newer underlying supported architectures when necessary. Apart from this $P4_{16}$ also brings new syntax and semantics to the P4 language making P4 programs more descriptive and feature rich. $P4_{16}$ implements PISA as one of the architecture as part of the PSA. We explore P4 using PISA in this thesis while using both $P4_{14}$ and $P4_{16}$ as our choice of DSLs. For the sake of brevity, we use the notation P4 to refer both $P4_{14}$ and $P4_{16}$ versions of the language while explicit reference to a specific P4 version is done only when necessary.

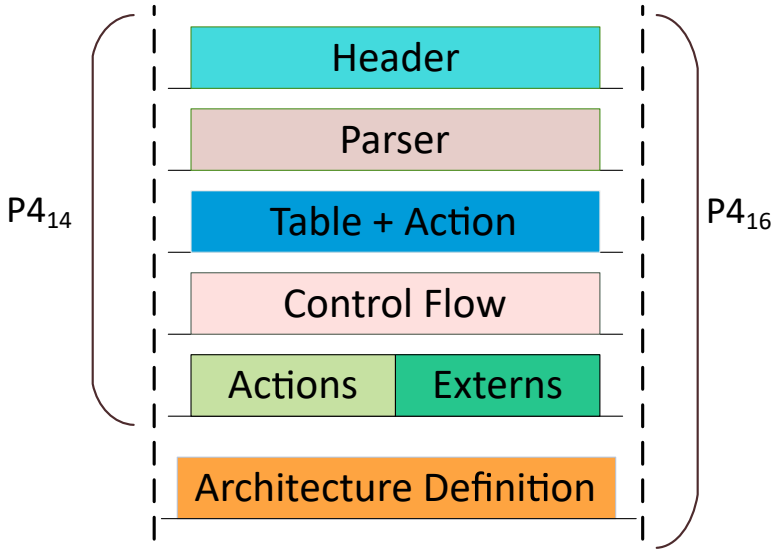


Figure 5 – Components of a P4 Program

We now bring our focus onto P4 programs written in bot $P4_{14}$ and $P4_{16}$. A P4 program consists of four crucial elements similar to the depiction in Figure 5 with two additional components available specifically for $P4_{16}$. The following description of the language components is based on $P4_{14}$ syntax and semantics. Although with $P4_{16}$ the syntax has been modified and language has become leaner with fewer keywords, the basic functionalities remain the same. The $P4_{16}$ related details can be referred at (BUDIU; DODD, 2017).

1. **Header Declaration**

   The P4 construct *Header* can be used to declare both header and metadata instances as metadata is identified as a special type of header. Header type specifies the associated fields and their widths of a header and normally sits at the beginning of a P4 program. An example of the Ethernet header, and a metadata are shown at Listing 2.1, and Listing 2.2 respectively. Metadata are declared per packet and remain valid until the packet goes out of the pipeline. It is necessary to create an instance of the headers and metadata after their declaration to enable reference to them while processing packets in the P4 program.

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

Listing 2.1 – Ethernet Header Definition

```
header_type local_metadata_t {
  fields {
    cpu_code        : 1;    // Code for packet going to CPU
    port_type       : 0;    // Inbound or Outbound Port
    ingress_error   : 1;    // An error in ingress port check
    is_tagged       : 0;    // If pkt is tagged
  }
}
metadata local_metadata_t local_metadata;
```

Listing 2.2 – Metadata Definition

2. **Parser Specification**

   Parser specification is usually the second part of any P4 program and always starts with ‘start’ parser state. It can be represented as a parse graph shown in Figure 6.

Parser Specification allows to parse an incoming packet in accordance to the headers declared in the P4 program (see Listing 2.3).

```
parser start {
  return parse_ethernet;
}

parser parse_ethernet {
  extract(ethernet);
  return select(latest.etherType) {
      0x0800 : parse_ipv4;
      0x86DD : parse_ipv6;
      default: ingress;
  }
}

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

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

Listing 2.3 – P4 parser example

We used *select* and *extract* statements in this parser example. Packet header details are retrieved using the *extract* statement and then the *select* statement determines the next protocol header to process based on the existing parsed data. The *select* statement also contains a reference to the Control Flows (e.g., ingress) used as the exit criteria for the Parser indicating the switch to commence processing the parsed packet data. The example shows that the parsing process starts with the Ethernet header and then reference the parsing function of the IPv4 or IPv6 protocol decided by the value of EtherType.

3. **Table and Action Definition**

The actions defined and the table declarations come after the parser specification in a P4 program. Table declaration is composed of *read* and *actions* components. *read* dictates the exact header fields to match upon and the table lookup algorithm to be used by the table. Likewise, *actions* specifies a sequence of actions available for the table. Each table entry is associated with an action to be enacted on the receiving packet should there be a match (table hit), otherwise in case of a table miss (no entry is matched) the default action for the table is referenced.

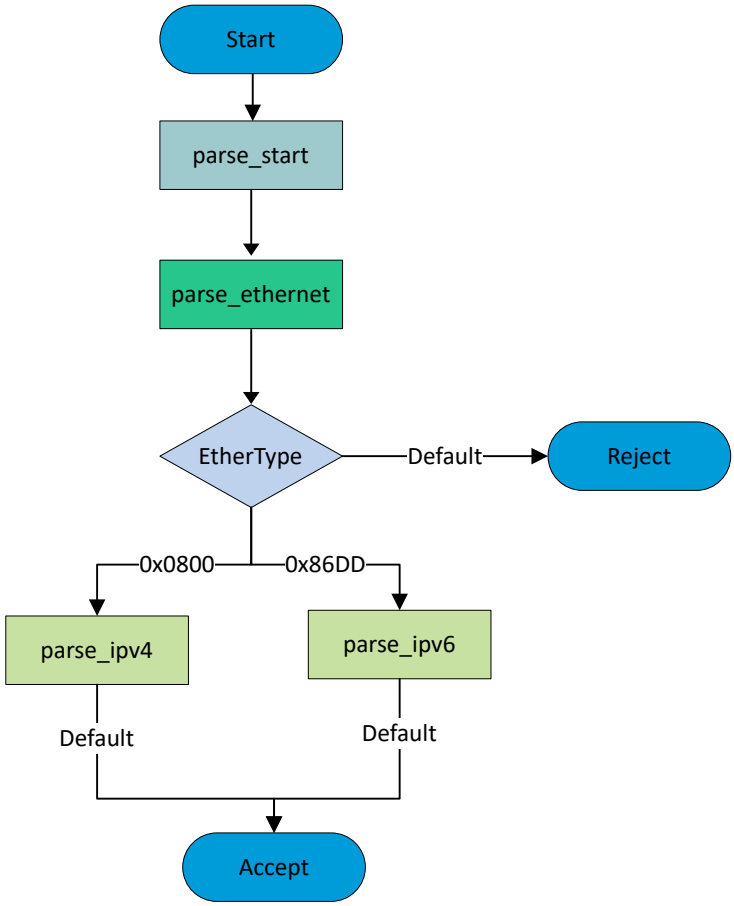


Figure 6 – P4 Parser Graph Example

```
action no_op() {
}

action rewrite_src_mac(smac) {                  //Compound Action
  modify_field(ethernet.srcAddr, smac);         //Primitive Action
}

table send_to {
  reads {
    standard_metadata.egress_port : exact;
  }
  actions {
    no_op;
    rewrite_src_mac;
  }
}
```

Listing 2.4 – P4 Match-action table specification example

The *actions* can be considered as functions which are built using Primitive Actions from the $P4_{14}$ specification and hence are also referred to as Compound Actions. Primitive Actions are a minimal set of instructions which can be used to describe

different simple packet processing actions such as *modify_field*, *add_header* etc. $P4_{16}$ has replaced the Primitive Actions with Extern functions which can be target specific primitive constructs for packet processing to bring more architectural support to the language. For backward compatibility, $P4_{16}$ provides support for all the Primitive Actions from $P4_{14}$ and also offers a tool to transform the $P4_{14}$ program into the $P4_{16}$ syntax.

4. **Control Flow**

   PISA dictates each network packet to be processed by a sequence of match+action tables. In a P4 program, this execution sequence is described by the Control Flow as shown in Listing 2.5. The tables are executed using the *apply* statement in the Control Flow. Table dependencies are enforced using *if-else* statements where the selection of the next table is decided as a result of the hit/miss outcome in the table under process.

```
1 control ingress {
2   apply(send_to);
3 }
```

Listing 2.5 – P4 control flow specification example

Full details and more examples about various elements of P4 language and of P4 programs can be found in P4 Specifications ($P4_{14}$ SPEC, 2017; $P4_{16}$ SPEC, 2017).

2.1.4 Control Plane API Abstractions

With SDN proliferation we now have the control plane as a separate logical entity from the data plane. It is trivial for the control plane to be able to control and manage the corresponding data plane for the switches to function correctly. This is where the control plane API Abstractions become essential for consideration. Projecting control plane as a separate centralized logical entity leads to a single control plane controlling a single or multiple data planes. Also, it is possible for the data planes to be of different architecture bringing more compatibility challenges to the control plane. We will explore some solutions which try to introduce abstractions to the control plane APIs making it functional over different data planes.

2.1.4.1 OpenFlow (OF)

OpenFlow (OF) is the front runner SDN technology bringing programmability to the data plane. It exposes new control knobs for programming the network, but the knobs' functions are largely dictated by the fixed functionality of the forwarding devices. The OF specification defines a switch pipeline by using multi-table matching to bring

flexibility to the data plane. Although it can add or remove flow entries in the table or define the sequence of tables to be part of the pipeline, it can not configure the width, depth or number of table of the underlying platform as the resources available are fixed in nature in the underlying platform. Likewise, OF also works with predefined protocols and header structures (e.g., Ethernet, IP, UDP) for parsing and matching. The protocol dependent pipeline is limited by the table actions supported too (e.g., forwarding, dropping, decrementing TTLs, pushing VLAN header) to process network packets. For new header format support or new packet operation support it is necessary to bring changes in OF specifications and the underlying hardware too. Due to fixed-function device limitations, In fact, it is not possible to configure every OF1.x defined pipelines and abstractions from the already available OF specifications over current ASICs offerings due to fixed-function nature of the device. In addition to that, OF tried to include support for more services year on year as shown in Table 3. The slow hardware development process could not keep up with the complexity of designing new OF features every year. Although it brings more flexibility on paper, it is practically not feasible to develop target hardware with the complete support of OF specification using current technologies while still remaining financially viable. Nevertheless, the contributions from the OF community are undeniably a significant driving factor bringing SDN to the mainstream.

Table 3 – Year-on-Year Evolution of OpenFlow

OF Version	**Release Date**	**Match Fields**	**Depth**	**Size (bits)**	**Major Features**
1.0	Dec 2009	12	12	264	Single Table. Ethernet/IPv4
1.1	Feb 2011	15	15	320	Multi table and Group table. VLAN and MPLS support.
1.2	Dec 2011	36	9-18	603	TLV matching. IPv6. Multiple Controller.
1.3	Jun 2012	40	9-22	701	Meter Table. MAC-in-MAC. Multiple Channel Between Switch and Controller.
1.4	Oct 2013	41	9-23	709	Synchronized Table. Bundle. Flow Monitoring.
1.5	Dec 2014	44	10-26	773	Egress Table. Schedule Bundle. Packet Type Aware Pipeline.

2.1.4.2 Switch Abstraction Interface (SAI)

Switch Abstraction Interface (SAI) (2014) presents a hardware abstraction model towards standard switch configurations APIs for switching silicon (ASICs). It represents the switch ASIC as a userspace software application. It is a set of standardized C language based APIs which a user can use to program the network hardware tables or configure

any network feature of the supported switch ASIC. As solely implemented in software, it allows developers to bring this solution over different Linux distributions and to port it to different switch ASICs by just changing the underlying SAI driver. With SAI a customer can configure and control the supported switching ASICs as described by the SAI specification. Having said that SAI is also limited by the fixed-function of the switches similar to OF and does not provide support for protocol independent programmable hardware. The SAI project has been adopted by the Open Compute Project (OCP) and seen acceptance from various switch silicon vendors in the networking industry like Cavium, Broadcom, etc.

2.1.4.3 Ethernet Switch Device Driver Model (switchdev)

In 2015, kernel networking developers adopted a new driver model called ethernet switch device driver model (switchdev) (ETHERNET. . . , 2014) that is aimed at replacing the proprietary blobs a.k.a SDK with standard kernel interfaces. As an in-kernel abstraction model, it keeps the switch state inside the kernel and works with the existing Linux applications instead of investing in the development of new tools. In time with more vendors making their drivers upstream, it can be a promising solution to break through the vendor's lock-in in network devices. *switchdev* already supports L2 data forwarding (switching) and L3 Routing Offload, and more features are being added continuously. Mellanox has already contributed to *switchdev* and have offered drivers for its Switch X-2 chip and also for its 100GB Spectrum chip. If more ASIC vendors will upstream their drivers and the NOS developers will integrate these changes, then *switchdev* can become a dominant solution towards open networking.

2.1.4.4 P4Runtime (PRT)

P4Runtime (PRT) is a vendor-independent and protocol-independent runtime API platform for P4-described data planes. Figure 7 represents the reference architecture of PRT. PRT is viewed in terms of PRT APIs, PRT Client, and PRT Server. PRT Server and PRT Client lie in the P4 target and the Controller respectively, whereas PRT APIs defines the runtime interface semantics between the target and controller. The standard messages and their format used are also described as part of the PRT APIs.

The P4Runtime API is specified by the p4runtime.proto protobuf file which is compiled by Protobuf compiler (protoc) to generate both server and client implementation stubs. While the controller maintainers instrument the client stubs, the data plane or target implementers instrument the server stubs. The primary workflow of PRT dictate the P4 program to be compiled to produce both P4 device specific *config* file and P4Info metadata responsible for forming the message format conjointly known as *ForwardingPipelineConfig* in PRT terminology upon which the communication between controller

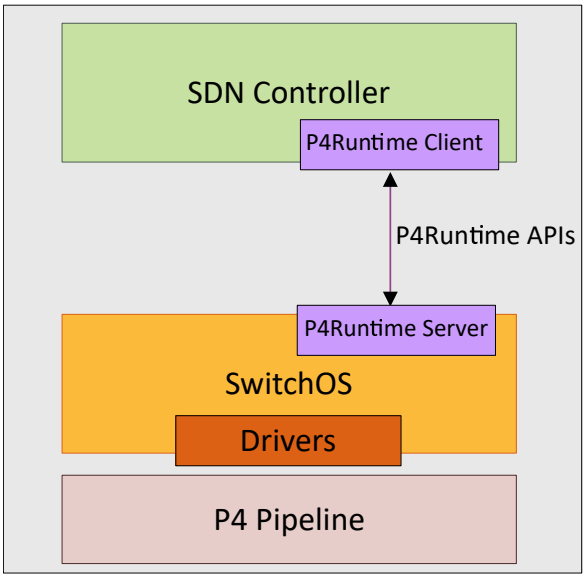


Figure 7 – P4Runtime Reference Architecture

and target will be based on. P4 Compiler also produces a P4Info Schema which is target and architecture independent. The P4Info schema includes the entity (i.e., P4 construct) instances from the P4 program (i.e., tables and extern instances). The entity instances are associated with a numeric ID assigned by the P4 compiler. The controller utilizes the P4Info schema and the target-specific device *config* file details to configure the P4 target. P4Runtime (PRT) is an ongoing activity and still under development without any reference implementation over a target. Although Google has demoed a working PRT implementation over its in-house NOS known as *Stratum*, it is still under incubation and has not been released for public. We believe once PRT comes out as a fully done solution, it will change the face of the SDN network bringing programmability to every aspect of a SDN network.

2.2 Related Work

We studied various works towards switch architectures and switch technologies. While some research works help us to understand and solidify our domain knowledge, others identified themselves to be a lot more influential to our research. We present here some of the related works which we understand have begun the journey towards flexibility of network in different ways and have contributed immensely intellectually to other contemporary and future research works. To begin with, Cuckooswitch (D. Zhou et al, 2013) is constructed around memory-efficient and highly-concurrent hash table design to achieve high throughput even with one billion flow entries. It is developed with a singular focus on high-speed table lookup and is optimized heavily going farther from generic design. On the contrary, the Click modular router is built over highly modular and configurable CLICK (KOHLER *et al.*, 2000) software architecture. It is constructed using configurable

packet processing modules such as packet classification, queuing, etc., composed into a directed graph to create a packet processing flexible router. Likewise, RouteBricks (M. Dobrescu et al, 2009), a CLICK based router, brings high performance by parallelizing router functionality across multiple cores and also across multiple servers. On the other hand while Packetshader (HAN *et al.*, 2010) achieves high performance by using GPU to avoid CPU bottleneck in software routers, Snap (SUN; RICCI, 2013) brings flexibility and configurability using CLICK modules to GPU based software routers. Although these switches provide programmability, configurability to an order or excel in achieving high performance, they lack the support of a high-level DSL i.e., P4.

On a similar note, OvS(2009) (PFAFF *et al.*, 2015) is an open source virtual switch with advance flow classification and caching techniques for improved performance and has been the spearhead in NFV developments. It comes in both user space and kernel space flavors. Being a Linux based switch, it runs on various environments including Virtual Machines (VMs), containers, etc. Although OvS achieves a higher degree of portability, the programmability under the hood is limited. Meanwhile, PISCES(2016) (M. Shahbaz et al, 2016) tries to bring P4 DSL support to OvS, but it is restricted by the OvS pipeline limitations to achieve protocol independence and only support a small set of P4 abstractions. Towards multi-platform support, Software for Open Networking in the Cloud (SONiC) (SOFTWARE. . . , 2016) form Microsoft is developed as an open switch OS with inherent SAI APIs feature support, but it lacks any DSL support. Whereas OpenSwitch (OPX) (2017) (OPENSWITCH. . . , 2017) open source multi-platform project from Linux-Foundation [1] brings limited P4 support towards protocol independence. Another project by Netronome (NETRONOME, 2015) supports the majority of P4 abstractions and brings protocol independence to its own proprietary Network Flow Processor (NFP) hardware. Table 4 shows different switch projects and their feature support to understand how different project groups target differently to the current requirements. We observed that the solutions available are either focused on performance, or programmability and flexibility but never target for all the characteristics. The open source projects provide very limited, or no DSL support with the only exception are in alignment with the proprietary solutions providing a near complete P4 support.

Then we have Translator for P4 Switches (T4P4S) (LAKI *et al.*, 2016; T4P4S. . . , 2016) which defines high performance data plane using P4 program as input. T4P4S compiler system implements a networking hardware abstraction layer (NetHAL) to support multiple platforms. The NetHAL brings target-dependent optimization while T4P4S compiler system also provides target-independent optimization as an integral feature of the compiler system. It implements the target dependent abstractions over Data Plane Development Kit (DPDK) APIs, e.g. Hash Table Lookup, IPv4 LPM Lookup, etc., as

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

Table 4 – Scope, Approach, and Feature Comparison List of different Programmable Switch Projects

Project	**Protocol Independent**	**Development Effort**	**DSL Support**	**Target**	**Remarks**
Click	Yes	Medium	No	General-Purpose Server	Mostly used for research
OVS	Limited	High	No	Software Switch	Runs as part of Linux kernel
Switchblade	No	High	No	FPGA	Verilog frontend
P4-Fpga	Limited	High	Yes	FPGA	Bluespec Compiler
P4-NetFpga	Yes	High	Yes	NetFPGA	Xilinx P4-SDNet tools
Cuckoo switch	Low	High	No	General-Purpose Server	CuckooHash based FIB Lookup
Packetshader	No	High	No	General-Purpose Server	GPU-assisted packet processing
Routebricks	No	High	No	General-Purpose Server	multi-core packet processing.
Pisces	Yes	Low	Limited	Software Switch	OVS Based
Routeflow	No	High	No	Openflow Device	Provides only control plane
T4P4S	Yes	Low	Yes	Limited by DPDK*	*Optimized for Intel
MACSAD	Yes	Low	Yes	Multi-Target	X86 & ARM support available

part of the NetHAL to define high-performance data plane. It showcases various use cases like L2 switching, L3 routing, Load Balancing, etc., but no IPv6 support though.

With a different approach built from the ground up Stratum [2] is an open source implementation for a thin switch targeting various white box switches. It is a silicon-independent switch operating system and can be managed by local (in case of Traditional switch design) or remote Network OS (NOS) using next-generation SDN interfaces such as P4Runtime and OpenConfig. It uses P4 to define logical data plane pipeline and uses P4Runtime to bring dynamic programmability to the pipeline in the switch. Although it is an open source project as part of Open Networking Foundation (ONF), as of now it is still in its incubation phase and has not released the source code to the public. Hence details about Stratum are sparsely available through press releases only.

We explained T4P4S and Stratum projects in little more details because they project a closer picture to our vision and what we want to achieve. Stratum can be a disruptor in the SDN world with its bold vision and supported features, but it is still in incubation phase and not available for testing and evaluation. Also, Stratum proposes an entirely new operating system which is not on our roadmap. On the other hand, T4P4S project shares a large part of our vision differing in terms of technology it uses for its compiler system. While T4P4S achieves higher performance built upon heavily tuned and customized DPDK, we choose ODP towards portability. We explored and evaluated T4P4S while comparing it against our proposal taking advantage of its similarity to our work. With our inclination towards Programmability, Portability, Performance, and Scalability (3PS) of data plane applications and careful selection of underlying technologies position our MACSAD proposal uniquely. We describe about MACSAD in detail in chapter 3.

[2] https://stratumproject.org

2.3 Concluding Remarks

This chapter summarizes the projects, tools, and technologies closely related or coinciding with this thesis. We present this chapter in two major sections describing Related Technologies and Related Works. Related Technologies explored the concurrent projects relevant to our thesis in different ways. Similarly, Related Works provides a mental picture of the existing projects and their features, and how they differ from our vision and our proposal.

Taking the cues from this chapter related to PISA, ODP and P4, we formulate our discussion in chapter 3, and we analyze how the missing features identified in the related works in section 2.2 are addressed.

3 Multi-Architecture Compiler System for Abstract Dataplanes

SDN brings a clear and programmatic separation between control and data plane functions by putting the control plane in a logically centralized location. Despite having received less attention compared to the control plane aspects of SDN, the data plane is a critical piece of every network switch. The OF protocol recognized the importance of data plane and provided a standard interface to the controllers to manage any OF compliant underlying data plane. Inherent inflexibility in OF deters data plane programmers to achieve higher flexibility. P4 being a descriptive programming language recognizes programming abstraction for network devices like header, parser, table, etc., which were made popular by OF, and provides language constructs for these higher abstractions. Now with P4 it is feasible to define a packet processing pipeline for a switch with programmable data plane.

On the other hand, ODP is another project attempting to bring platform-agnostic SDKs for switch datapath chips. We bring the Top-Down approach of P4 towards programmability and Bottom-Up approach of ODP towards portability together to propose our research work Multi-Architecture Compiler System for Abstract Dataplanes (MACSAD). Thus and so MACSAD (PATRA *et al.*, 2016; PATRA *et al.*, 2017; PATRA *et al.*, 2018) is created aiming to hide data plane programming complexity using P4 while keeping the flexible data plane portable, and scalable through the performance and hardware acceleration features of ODP. From an implementation aspect, it merges protocol-independent P4 abstractions and primitives with ODP APIs towards data plane applications.

3.1 Architecture

The high-level architecture of MACSAD (see in Figure 8) embodies three separate modules in sought for 'Protocol Independence' and 'Target Independence'. We explain the different modules in detail followed by how they contribute towards different features of MACSAD. The three modules are:

Auxiliary Frontend: Supports different frontend DSLs with $P4_{14}$ and $P4_{16}$ support in place.

Auxiliary Backend: Implements DSL abstractions over target-agnostic ODP APIs to support different platforms.

Core Compiler: Composed of 'Transpiler' and 'Compiler' submodules to create any data plane application.

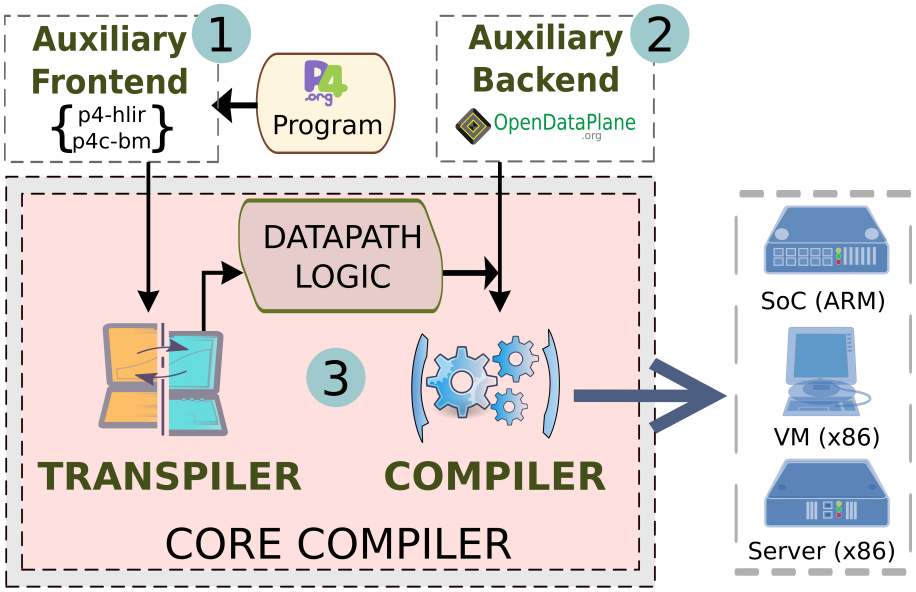


Figure 8 – High-level Reference Architecture & Use Case Workflow.

3.1.1 Auxiliary Frontend

Auxiliary Frontend transforms a P4 program into an Intermediate Representation (IR) suitable for the *Core Compiler* module by integrating projects from P4 consortium. Incorporating the ***p4-hlir*** project, *Auxiliary Frontend* translates $P4_{14}$ programs into High Level Intermediate Representation (HLIR) (P4..., 2018) format. HLIR is an in-memory abstract syntax tree (AST) data structure which can represent the P4 program as a python data structure to be consumed by a compatible P4 compiler. By creating an independent module for *Auxiliary Frontend*, MACSAD eases the effort to add support for newer DSLs in future simply by implementing new or extending the current code for the new DSL without affecting other modules of MACSAD. As a result, we are able to add support for $P4_{16}$ to *Auxiliary Frontend* with minimal changes to the existing code. A JavaScript Object Notation (JSON) IR is created for $P4_{16}$ program by integrating `p4c-bm`[1] project, also from P4 consortium. The top left rectangle in Fig. 8 depicts the transformation of P4 program into an unambiguous IR before being passed on to the *Transpiler* submodule, part of the *Core* module.

3.1.2 Auxiliary Backend

As the name suggests, *Auxiliary Backend* is responsible for backend or target related components of MACSAD. *Auxiliary Backend* comprises of all internal and helper

[1] https://github.com/p4lang/p4c-bm

APIs of MACSAD, and implements them over ODP APIs to support P4 abstractions turning MACSAD into a unifying compiler system achieved via the common SDK a.k.a. ODP APIs. From a classical compiler perspective, *Auxiliary Backend* can be considered as an auxiliary library. P4 abstractions are much higher compared to the abstraction understood by ODP, and they can not be mapped one-to-one. *Auxiliary Backend* bridges this gap to implement necessary APIs needed to define a data plane. These APIs span resource handling, CPU core management, table management, port configuration, packet manipulation, Packet I/O, controller support, etc. We coarsely categorize the APIs in relation to the P4 abstractions appeared in the P4 program as Helper APIs whereas the other APIs are considered to be MACSAD Internal APIs. The APIs related to CPU core management, Packet Tx/Rx, Fast Packet Processing Abstraction (DPDK, Netmap related), remote Controller support, etc., are part of the Internal APIs whereas header manipulation, table management, etc., are considered to be Helper APIs section 3.4. *Auxiliary Backend* also abstracts the hardware acceleration features (such as Crypto) allowing a developer to write applications while being unaware of the nuances of the target platform and their SDKs. All the components offered by *Auxiliary Backend* can be broadly categorized into Target-Independent and Target-Dependent APIs as shown in Table 5, and are explained in this chapter. Adding support of a new target platform for MACSAD is equivalent to porting only the ODP, or to be specific ODP APIs, onto the new platform.

Table 5 – Packet Processing Functions

Target-Independent	**Target-Dependent**
(add, remove, copy)_header, generate_digest, modify_field, Table Configuration, Header Parsing	push, pop, count, meter, Pkt Rx/Tx, Checksum, Table Creation, Table Lookup

3.1.3 Core Compiler

Core Compiler is the heart of MACSAD and encompasses the *Transpiler* and *Compiler* submodules. It compiles the IR received from the *Auxiliary Frontend* along with the APIs provided by the *Auxiliary Backend* into MacS (hereafter, the MACSAD compiled binary code is referred to as MacS throughout the text) for the desired target platform.

3.1.3.1 Transpiler

The MACSAD code base comprises of two categories of code (1) *Static Code* written over Internal and Helper APIs and are an essential part of the Auxiliary Backend (see subsection 3.1.2). (2) *Auto Generated* code with internal references to the Helper APIs to bring P4 abstractions into fruition. The *Transpiler* submodule is our template

based source-to-source compiler solution to output the auto-generated code written in 'C' from the P4 program. The *Transpiler* submodule itself is developed in Python programming language. *Transpiler* takes in the *Auxiliary Frontend* subsection 3.1.1 IR output to auto-generate a big chunk of MACSAD code.

Transpiler is responsible to map P4 components to the PISA architecture and generates the corresponding code for it. The auto generated code consists of the *Packet Parsing Logic* and the *Control Logic* for 'Programmable parser' and 'Match+Action' of PISA. The *Packet Parsing Logic* includes data structures for 'header fields, their offset & bitmasks' and handles packet header parsing. Similarly, the *Control Logic* expresses the packet flow across the tables defined in the P4 program. The *Control Logic* implements the target-independent functions (see Table 5). Together these two form the 'Data Path Logic' section 3.5 of MACSAD representing the P4 defined data plane in this compiler system.

The *Transpiler* performs the following actions during the source-to-source compilation:

1. Defines table constructs (e.g., size, lookup algorithm).
2. Generates the 'Data path Logic' code based on the IR from 3.1.1.
3. Maps loosely-typed DSL (i.e., P4[2]) to strongly-typed (i.e., 'C') declarations for auto generated *Data path Logic* code by selecting appropriate data types depending on the target platform.
4. Takes performance optimization decisions (e.g., RX burst size) based on predefined platform specificities.

3.1.3.2 Compiler

The *Compiler* submodule sits at the final stage of MACSAD and brings together all the different modules of MACSAD and the P4 program. It is responsible for the binary code generation of MacS using *Auxiliary Backend* (subsection 3.1.2) and output of *Transpiler* (subsubsection 3.1.3.1) over ODP APIs with the underlying GNU Compiler Collection (GCC) / Low Level Virtual Machine (LLVM) compilers. It brings the regular array of optimization tools supported by the underlying compiler benefiting the programmers.

[2] Due to providing high-level abstractions and fundamentally one type of variable, we considered P4 as a loosely-typed language

3.2 Compilation Process

MACSAD compiler system follows a three-step compilation process from P4 programs to MacS. Each of the MACSAD modules contributes to the steps of the compilation process as and when necessary. Generally, almost every compiler design support a frontend module which creates an internal IR for the input program as part of the initial step of compilation. This IR allows the compiler to apply various optimization techniques on it before compiling it to create the final target binary or platform image. MACSAD compilation process begins with *Auxiliary Frontend* taking a $P4_{14}$ or $P4_{16}$ program as input as the first step. Figure 9 summarizes the three steps of compilation visually. Step 1 creates outputs in HLIR or JSON IR format for $P4_{14}$ and $P4_{16}$ programs respectively, and bring the process handle from P4 language to Python/JSON IR. The details of these IRs are described in section 3.3. In Step 2, the just created IR output is passed to *Transpiler* submodule for auto-generation of Data Path Logic code. This code auto-generation is explored in detail in section 3.5. *Transpiler* is developed as a template based source code generator which transforms the input IR for P4 program into a set of 'c' and 'h' files to be consumed by the following *Compiler* submodule. With this Step 2, MACSAD compiler moves from python territory to 'c' language arena. Entering into Step 3, MACSAD utilizes *Auxiliary Backend* module subsection 3.1.2 and *Compiler* submodule subsubsection 3.1.3.2 along with ODP APIs, and output of 3.1.3.1 a.k.a. Datapath Logic Code to generate MacS.

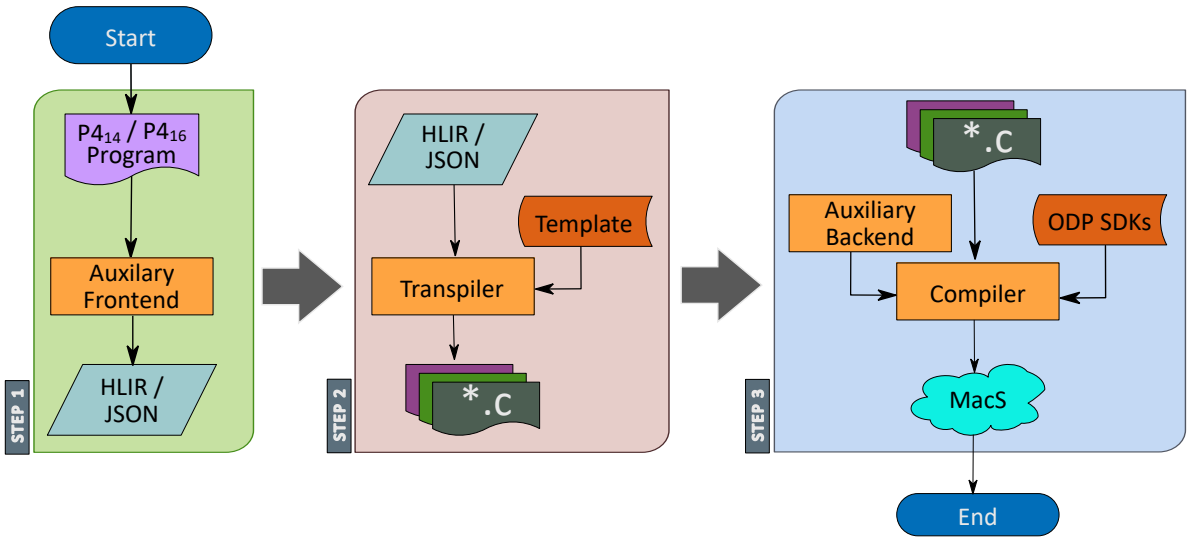


Figure 9 – Three Step Compilation Process.

3.3 P4 to IR Code Generation

As the first step of compilation, MACSAD begins with *Auxiliary Frontend* taking a $P4_{14}$ or $P4_{16}$ program as input and creates HLIR or JSON IR as output respectively before feeding them to the *Transpiler* submodule (subsubsection 3.1.3.1).

$P4_{14}$. According to P4 guidelines, the $P4_{14}$ program should be converted into HLIR IR (P4..., 2018) format by compiler frontend to represent the P4 program unambiguously. This HLIR is then used by the compiler backend to create the P4 data plane application. HLIR IR is a non-file based in-memory IR developed using python data structures. It can only be created and accessed during the compilation process. HLIR confirms to all semantic rules to represent P4 constructs established by the P4 specification, and can represent functionalities of any P4 program in its entirety. For every compiler, HLIR is the common entry point, and MACSAD adheres to this requirement.

Auxiliary Frontend builds the HLIR from the P4 program by integrating ***p4-hlir*** with the help of a small python code snippet as shown in Listing 3.1. It is necessary for ***p4-hlir*** open-source project from Github repository of P4.org to be installed in the system for completing this phase.

```
1 from p4_hlir.main import HLIR
2
3 h = HLIR(<list of p4 sources>)
4 h.build()
5
6 for table_name, table in h.p4_tables.items():
7     pass
```

Listing 3.1 – p4-hlir Integration Code Snippet

Every P4 abstraction type (e.g. table) is defined as a python class (e.g. p4_table) in HLIR . The P4 objects are represented as OrderedDict python data type and available as attributes of HLIR object. Listing 3.1 shows an example to access the tables defined in the P4 program by accessing the OrderedDict *p4_tables*. Table 6 shows a complete list of names of P4 objects available under HLIR.

$P4_{16}$. With the new version, P4 has taken a new direction in the development of the language. With its reference P4 switch **bmv2**[3], it moves towards a file-based IR developed in JSON format. MACSAD *Frontend* module adds support for $P4_{16}$ to be able to transform a $P4_{16}$ program into its JSON representation. ***p4c-bm*** [4] is the prerequisite tool for converting $P4_{16}$ to JSON format. The basic way to generate JSON file

[3] https://github.com/p4lang/behavioral-model
[4] https://github.com/p4lang/p4c-bm

is shown in Listing 3.2 and exists as an internal part of the source code of MACSAD *Frontend* module.

```
p4c-bm --json <path to JSON file> <path to P4 file>
```

Listing 3.2 – JSON IR Generation

Listing 3.3 shows a code snippet of the *ingress* control flow of a $P4_{16}$ program depicting the *sendout* table and two different action functions *on_miss* and *rewrite_src_mac*. The JSON format for this P4 snippet generated with the help of ***p4c-bm*** is presented in Listing 3.4. The JSON representation clearly interprets the P4 constructs and present them as different JSON objects where all the attributes of the P4 constructs are also presented clearly labeled with separate key and their values. JSON representation also includes a key *ID* to provide a unique identification for each P4 construct across the P4 program.

Keeping the current HLIR support active and useful, MACSAD Frontend takes an extra step and transforms the JSON representation of $P4_{16}$ into the HLIR representation before passing it to the Step 2 of the compilation process. This allowed us to add the new P4 version support with limited modification to the MACSAD code.

Table 6 – P4 Object List in HLIR

OrderedDict's name	**P4 object type**
p4_actions	action
p4_tables	table
p4_conditional_nodes	None, this is used to represent conditions in the control flow
p4_action_profiles	action_profile
p4_action_selectors	action_selector
p4_headers	header_type
p4_header_instances	header
p4_fields	None, this is used to refer to a field in a
p4_field_lists	field_list
p4_parse_states	parser
p4_parse_value_sets	parser_value_set
p4_parser_exceptions	parser_exception
p4_counters	counter
p4_meters	meter
p4_registers	register
p4_field_list_calculations	field_list_calculation

```
control ingress(inout headers hdr,
                inout metadata meta,
                inout standard_metadata_t
                standard_metadata) {

  @name(".on_miss")
    action on_miss() {}

  @name(".rewrite_src_mac")
    action rewrite_src_mac(bit<48> smac)
   {
    hdr.ethernet.srcAddr = smac; }

  @name(".sendout")
    table sendout {
      actions = {
        on_miss;
        rewrite_src_mac; }
      key = {standard_metadata.
   egress_port: exact; }
        size = 512; }
    apply { sendout.apply(); } }
```

Listing 3.3 – $P4_{16}$ Control Flow

```
"actions": [
{ "name": "on_miss",
  "id": 0,
  "runtime_data": [],
  "primitives": [] },
{ "name": "rewrite_src_mac",
  "id": 1,
  "runtime_data": [
   { "name": "smac",
     "bitwidth": 48 } ],
     "primitives": [
     { "op": "modify_field",
       "parameters": [
       { "type": "field",
         "value": [
          "ethernet",
          "srcAddr" ] },
          { "type": "runtime_data
    ",
            "value": 0
} ] } ] }
],

"pipelines": [
{ "name": "ingress",
  "tables": [
   { "name": "sendout",
    "id": 1,
    "key": [ {
      "match_type": "exact",
      "target": [
        "standard_metadata",
        "egress_port"
                    ], } ],
  "actions": [
    "on_miss",
    "rewrite_src_mac" ],
  "next_tables": {
    "on_miss": null,
    "rewrite_src_mac": null },
    "base_default_next": null
} ], }
]
```

Listing 3.4 – Control Flow in JSON Format

3.4 Internal & Helper APIs

MACSAD features and functionalities are implemented using a number of APIs designed as part of the *Auxiliary Backend* 3.1.2 module. These APIs bind together the auto-generated code to the MACSAD APIs to bring high-level P4 program onto the given low-level target platform. These APIs being an integral part of *Auxiliary Backend* helps the *Compiler* submodule in the compilation process, and also in creating the final MacS for the target. Helper APIs are self-explanatory and explore the Parser and Table functionalities of a P4 program providing the APIs to implement these features over ODP SDKs to support the bottom-up effort of ODP. We identify all the APIs which can be referenced from the *Transpiler* auto-generated code as Helper APIs. Rest of the APIs from *Auxiliary Backend* are refereed as Internal APIs. Internal APIs work towards connecting the components of the switch software may it be plumbing internal modules or managing session towards an external controller. Hence it can be considered to be more of system level APIs necessary for MacS to work properly. This division is solely based on where the APIs are consumed internally in the MACSAD system. For brevity, we refer both types of APIs as Backend APIs in this text unless specified explicitly. Backend APIs are an amalgamation of different categories each consisting of multiple APIs. The categories span over system related, parser related, table related, and control plane related APIs. The selective list of APIs mapped to their categories are shown in the Table 7.

Table 7 – Backend APIs Categorical Examples

API Category	**API List**
System	creatCtrlSession, destroyCtrlSession
	createPktio, destroyPktio
	createThread, startThread
Parser	addHeader, copyHedaer, removeHeader
	updtPktField, getPktField
	getByteOffset, getByteWidth
Table	addExactEntry, delExactEntry
	addLpmEntry, delLpmEntry
	addLpm6Entry, delLpm6Entry
	getExactEntry, getLpmEntry, getLpm6Entry

3.5 Source to Source Code Transformation

Source to Source code transformation auto generates the packet processing code for MACSAD which turn out to be a big part of the MACSAD code base. It begins with generating the HLIR IR from P4 program followed by another code transformation phase courtesy of *Transpiler* submodule. MACSAD brings the P4 abstractions to the level of ODP abstractions only to be realized by transforming P4 into 'C' language on which ODP is based on. While HLIR IR is the intermediate language, it gets transformed into a set of '.c' and '.h' files based on the 'C' language. Undoubtedly this auto-generation of code entails a lot of attention and is explained in detail here. As mentioned in subsubsection 3.1.3.1, the auto-generated code is referred to as Data path Logic code which is comprised of two code sets targeting a different aspect of packet processing pipeline, in our case specifically PISA architecture. While Packet Parsing Logic targets the Programmable Parser, Control Logic describes the packet flow across the tables and corresponding actions necessary to be enforced.

Although the IR from *Auxiliary Frontend* is an unambiguous representation of the P4 program, it is an in-memory and non-file based representation using Python language or in JSON format for $P4_{14}$ and $P4_{16}$ respectively. The P4 objects are represented as Key+value pair which is a distinct characteristic of descriptive languages such as JSON, YAML, etc. It is a loosely typed representation where data types are not explicitly defined. This type system provides a lot of flexibility, but also difficult to debug in case of error as the compiler cannot enforce stricter rules to check the data types. Meanwhile, ODP is a 'C' based project with its APIs, and abstract data types conform to 'C' language. The only way to bring P4 and ODP together is to transform the IR into 'C' based code. The under the hood compatible compilers of choice, which are GCC and LLVM, also requires the P4 code to be presented in 'C' language format. *Transpiler* submodule is developed inline to this requirement bringing P4 to the strongly-typed low-level language. The remainder of this section explores the requirements, faced challenges and the process for this code transformation.

Every P4 program has two logical section where one describes the data types and data structures, while the other focuses on the functionality part of the program. While doing a code transformation, we also focus on both these sections separately. This allows us to bring improvements to both the sections separately while maintaining the synergistic collaboration between them. For the purpose of *Transpiler*, we identified P4 core language constructs into 6 different P4 language abstractions. All the auto-generated code are spread across six sets of '.c' files and '.h' files each corresponding to P4 abstractions mentioned in the Table 8. We maintain a template file for each file set (*c, *h) generated as a result of code transformation by the *Transpiler*. The *Transpiler* and the template files are implemented in Python to facilitate working with HLIR IR, also Python based.

The complete set of auto-generated files are mentioned in Listing 3.5.

```
1  [*.h]->    parser.h    actions.h    pipeline_data.h
2  [*.c]->    parser.c    actions.c    dataplane.c    tables.c
    controlplane.c
```

Listing 3.5 – Auto Generated File List

Table 8 – Transformation of P4 Constructs to 'C' Language

P4 Abstraction	**Auto Generated Files**	**Remarks**
Headers	parser.h actions.h	Describe the format (the set of fields and their sizes) of each header within a packet.
Parser	parser.c	Describe the permitted header sequences within received packet as a finite-state machine.
Tables	tables.c pipeline_data.h	Associate look-up keys to actions. P4 tables generalize traditional forwarding tables; they can be used to implement routing tables, flow lookup tables, access-control lists, etc.
Actions	action.(c,h)	Describe how packet header fields and metadatas are configured.
Match-action	dataplane.c	Stitch together tables and actions, and perform the following sequence of operations: Construct lookup keys from packet fields or computed metadata; Use the constructed lookup key to index into tables, choosing an action to execute; Finally, execute the selected action.
Control Flow	controlplane.c	Expressed as an imperative program describing the data-dependent packet processing within a pipeline, including the data-dependent sequence of match-action unit invocations.

This complex code transformation is carried out in two phases. First *Transpiler* reads the details of different P4 constructs from HLIR and create a python (.py) file for the P4 constructs using the appropriate Template file as reference. For each P4 construct, we have one template file to take part in the code transformation. *Transpiler* identifies the specific components for each P4 construct defined in the template file and creates the intermediate python file in the process. This python file is executed in *Transpiler* using the underlying python compiler to generate the final 'C' based source including the header files. Primary code flow for generating code for P4 table abstraction is shown in Figure 10.

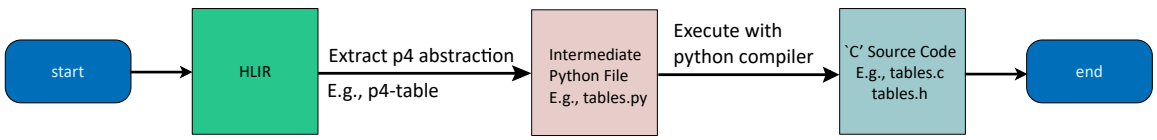


Figure 10 – Code-autogeneration Flow Diagram.

The followed discussion about code generation focuses on the initial transformation of P4 abstractions to low-level language explaining the transformation process in detail in subsection 3.5.1 producing all the final data types and data structures generated as the outcome of this phase. Then we explain the final code generated satisfying the functionality of the P4 program in subsection 3.5.2. This explains the Parsing Logic and Control Logic of the P4 program, and how it is represented in the 'C' language after the transformation. As part of the transformation, *Transpiler* saves the intermediate python programs for back reference purposes useful for debugging when necessary.

3.5.1 Transforming Language Abstractions

P4 constructs and the HLIR objects (see Table 6) are very high-level abstractions which do not have any direct corresponding data types in 'C' language, not even in ODP level data types which sits above the 'C' based abstractions. For a source-to-source transformation, we need to identify the high-level data types to be transformed of the initial language, and also the low-level data types of the final language which are rich enough to represent the high-level data types in some way.

For example, the value of a P4 field is a number; it can be represented as a data type of different lengths like short, int, long, etc., in C language. Similarly, a Header type in P4 may be represented as a complex data structure such as "struct" or "enum", unlike other basic 'C' data types which are not rich enough to represent these complex abstractions. Hence the *Transpiler* needs to make a number of intelligent decisions while transforming loosely-typed P4 abstractions into strongly-typed low-level simple data types or complex data structures. In a P4 program, the data types appear either as header fields or as parameters to action functions.

Packet Header is indeed a critical abstraction to consider in this transformation process as the whole packet processing pipeline starts and ends with a network packet. Unlike other projects viz. *p4-ebpf* [5], p4 headers are not transformed into a struct data type in MACSAD. Instead, we try to direct our focus to the different features and properties of the P4 abstraction and create multiple data structures to work collectively. This allows better inlining of code snippets towards better performance.

A simple Ethernet header in P4 shown in Listing 2.1 is expressed using an enum and two arrays when transformed into 'C' language. Hence all the headers and metadata can be represented using these fixed number of enums & arrays irrespective of the header counts. This allows name/index based reference while keeping the number of data structures in check. Table 9 and Table 10 show the transformation of header instance level features and header field level features respectively into corresponding enums and arrays. The enums and arrays defined will keep adding more values to the data types as

[5] https://github.com/p4lang/p4c/tree/master/backends/ebpf

per the header counts present in the P4 program. Table 9 shows that for identification of header and metadata instances, an enum is created which assigns an integer value to each instance. Due to the inherent features of enum, we can reference the instances with the assigned integer value or using their name without worrying any string comparison function. The header_instance_byte_width array is populated with the header lengths, and the index relates to the values from header_instance_e enum defined for header instances. Similarly, header_instance_is_metadata array shows if a header is a metadata type or not. Table 10 also follows the same steps to represent the header fields by starting with an enum for the field instances, and then followed by different arrays focusing on the field attributes like width, byte offset, etc.

Table 9 – Auto-generated Code for Header Instances

```
enum header_instance_e {
 header_standard_metadata,
 header_ethernet
};
```

```
static const int header_byte_width[HEADER_INSTANCE_COUNT] = {
  20 /* header_standard_metadata */,
  14 /* header_ethernet */
};
```

```
static const int header_is_metadata[HEADER_INSTANCE_COUNT] = {
  1 /* header_standard_metadata */,
  0 /* header_ethernet */
};
```

Table 10 – Auto-generated Code for Header Field Instances

```
enum field_instance_e {
  standard_metadata_ingress_port,
  standard_metadata_packet_length,
  standard_metadata_egress_spec,
  standard_metadata_egress_port,
  standard_metadata_egress_instance,
  standard_metadata_instance_type,
  standard_metadata_clone_spec,
  standard_metadata__padding,
  ethernet_dstAddr,
  ethernet_srcAddr,
  ethernet_etherType
};
```

Table 10 continued from previous page

```
static const int field_instance_bit_width[FIELD_INSTANCE_COUNT] = {
  9 /* standard_metadata_ingress_port */,
  32 /* standard_metadata_packet_length */,
  9 /* standard_metadata_egress_spec */,
  9 /* standard_metadata_egress_port */,
  32 /* standard_metadata_egress_instance */,
  32 /* standard_metadata_instance_type */,
  32 /* standard_metadata_clone_spec */,
  5 /* standard_metadata__padding */,
  48 /* ethernet_dstAddr */,
  48 /* ethernet_srcAddr */,
  16 /* ethernet_etherType */
};
```

```
static const int field_instance_bit_offset[FIELD_INSTANCE_COUNT] = {
  0 /* standard_metadata_ingress_port */,
  1 /* standard_metadata_packet_length */,
  1 /* standard_metadata_egress_spec */,
  2 /* standard_metadata_egress_port */,
  3 /* standard_metadata_egress_instance */,
  3 /* standard_metadata_instance_type */,
  3 /* standard_metadata_clone_spec */,
  3 /* standard_metadata__padding */,
  0 /* ethernet_dstAddr */,
  0 /* ethernet_srcAddr */,
  0 /* ethernet_etherType */
};
```

```
static const int field_instance_byte_offset_hdr[FIELD_INSTANCE_COUNT] = {
  0 /* standard_metadata_ingress_port */,
  1 /* standard_metadata_packet_length */,
  5 /* standard_metadata_egress_spec */,
  6 /* standard_metadata_egress_port */,
  7 /* standard_metadata_egress_instance */,
  11 /* standard_metadata_instance_type */,
  15 /* standard_metadata_clone_spec */,
  19 /* standard_metadata__padding */,
  0 /* ethernet_dstAddr */,
  6 /* ethernet_srcAddr */,
  12 /* ethernet_etherType */
};
```

Table 10 continued from previous page

```
static const header_instance_t field_instance_header[FIELD_INSTANCE_COUNT]
  = {
  header_standard_metadata /* standard_metadata_ingress_port */,
  header_standard_metadata /* standard_metadata_packet_length */,
  header_standard_metadata /* standard_metadata_egress_spec */,
  header_standard_metadata /* standard_metadata_egress_port */,
  header_standard_metadata /* standard_metadata_egress_instance */,
  header_standard_metadata /* standard_metadata_instance_type */,
  header_standard_metadata /* standard_metadata_clone_spec */,
  header_standard_metadata /* standard_metadata__padding */,
  header_ethernet          /* ethernet_dstAddr */,
  header_ethernet          /* ethernet_srcAddr */,
  header_ethernet          /* ethernet_etherType */
};
```

Similar to header abstractions, the parameters to action functions are also need to be converted to low-level data types. In MACSAD, this is achieved using 'C' based structures, unlike the case of headers where enums and arrays were sufficient. Every function parameter is converted to an array of byte-sized data type of length equal to the byte width of the function parameter as shown here.

P4FIELD(name, length) is represented as uint8_t name[(length + 7) / 8]

For example, the Egress port is defined as 9 bit in P4 specification and will be converted into an array of 2 Bytes; likewise, mac address of 48 bits will be converted to an array of 8 Bytes. Although not apparent, at times *Transpiler* takes some compile-time decisions concerning the underlying target resulting in a different way of conversion. For example, the *Transpiler* decides to consider 1 Byte or 2 Bytes data types (such as uint8_t or uint16_t) to represent a port as appropriate to the target instead of a generic array type as done for mac address. Similarly, in various other cases, the *Transpiler* takes the rein and figure out the best data types available at the target, and auto-generates the code accordingly. We pack every parameter of an action function into a unique structure referred to as '<action_name>_params'. Then we have different structures for each table which incorporate all the structures of action parameters to form a Union data structure adding an action ID variable to it. Each instance of this '<table_name>_action' structure can specify a distinct action parameter structure using the actionID. Listing 3.14 and Listing 3.15 depicts the parameters of a P4 action function, and the corresponding transformed low-level code achieved by the *Transpiler* respectively. This transformation phase results in producing all the data types and data structures required to represent the P4 constructs present in the P4 program.

```
action rewrite_src_mac(smac) {                  //Compound Action
    modify_field(ethernet.srcAddr, smac);        //Primitive Action
}
```

Listing 3.14 – P4 Action Function Example

```
#define P4FIELD(name, length) uint8_t name[(length + 7) / 8];

enum actions {  // Enum for all the actions of P4 program
    on_miss,
    rewrite_src_mac,
};

struct rewrite_src_mac_params { // All parameters of an action
    P4FIELD(smac, 48);
};

struct sendout_action { // "sendout" Table
    int actionId;
    union {
        struct rewrite_src_mac_params rewrite_src_mac_params;
    };
};
```

Listing 3.15 – Auto-generated Code for Action Function

3.5.2 Auto-generating Data Path Logic

Addressing the P4 constructs from lower to higher complexity, we move our focus to more complex abstractions compared to P4 Field or Action function parameters. These include Parsers, Match+Action or Tables, and Control Flow, otherwise considered as the functional components of a data plane. This code transformation phase uses the auto-generated variables and data structures explained in subsection 3.5.1. The Internal and Helper APIs of Auxiliary Backend module are referenced heavily while transforming the functional components of the P4 program.

Parser block in a P4 program begins with the *start* parser state and continues parsing the headers in the sequence it appears in the packet. P4 uses *select* statement to compare against header field values to choose the next header for parsing as depicted in Listing 3.16. The Listing shows a snippet of a P4 program parsing an Ethernet header to start with, and then proceeding to parse IPv4 header after verifying its presence by checking the EtherType value in the already parsed Ethernet header. The transformed code present the same sequence of events as seen in the Listing 3.17. Each Parse function starts with extracting the header offsets and storing the pointers to the headers for future use in the pipeline code. Then it creates the key, i.e. the header field used in *select*

statement to choose the next parser function. Finally, *key* is compared, and based on it the next header to be parsed is decided. The code snippet shows how the Ethernet and IPv4 headers are parsed in sequence and how the *select* statement is implemented in auto-generated code.

```
parser start {
  return parse_ethernet;
}

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

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

Listing 3.16 – P4 Parser Code Snippet

```
static void parse_start(packet_descriptor* pkt) {
  return parse_ethernet(pkt);
}

static void parse_ethernet(packet_descriptor* pkt) {
  extract_header_ethernet(pkt);
  create_select_key(pkt, key);
  if (compare_key (key, 0x0800))
    return parse_ipv4(pkt);
}

static void parse_ipv4(packet_descriptor* pkt) {
  extract_header_ipv4(pkt);
  return ipv4_table1 (pkt);
}
```

Listing 3.17 – Auto-generated Parser Code

Following on we take Table, or to be exact Match+Action P4 abstraction, into consideration to explain its code transformation. Every Table construct has *reads* statement which dictate the *key* field for table lookup and the lookup up algorithm to use, and *action* functions listing all the actions applicable to the table. In addition to that the P4 program also defines the *action* functions which accept a list of parameters as function arguments while its body is described with P4 primitive actions or expressions updating headers and

metadatas. A snippet of the P4 program showing the Table **sendout** and the associated action functions is presented in Listing 3.18 where as the corresponding auto-generated code from *Transpiler* is shown in Listing 3.19. Table abstraction is implemented as a function which starts with constructing the lookup key followed by performing the actual table lookup. Then the lookup result, the return value of table lookup, which contains the action function parameters, a.k.a. action parameters, and the actionID are retrieved. ActionID identifies the associated action function to reference for the current packet, and the function arguments are passed to the action function. We implement the code to choose the proper action function using a switch 'C' construct. MACSAD Backend provides the APIs implementing the functionalities of the P4 Primitive Actions. Inside each action function code, the Helper APIs from *Auxiliary Backend* are referenced to act upon the packet headers or metadata. Here in this example, the modify_field primitive action is implemented as pkt_field_updt function in MACSAD while getByteOffset and getByteWidth are the functions implemented to retrieve details related to the header or metadata fields. The P4 Control Flow logic is added to the transformed code for Table abstraction. The code in Listing 3.19 shows how actionID is used in 'table_sendout' to choose the next table to be referenced.

```
action no_op() {}     //No Operation

action rewrite_src_mac(smac) {              //Compound Action
  modify_field(ethernet.srcAddr, smac);     //Primitive Action
}

table sendout {
  reads {
    standard_metadata.egress_port : exact;
  }
  actions {
    no_op;              //On Miss
    rewrite_src_mac;    //On Hit
  }
}
```

Listing 3.18 – P4 Table Code Snippet

```
void no_op(packet_descriptor* pkt) {}

void rewrite_src_mac(packet_descriptor* pkt, lookup_table** tables,
                     struct rewrite_src_mac_params parameters) {
    updtPktField(pkt, getByteOffset(ethernet_srcAddr),
                    getBytewidth(ethernet_srcAddr));
}

void table_sendout(packet_descriptor* pkt) {
```

```
10      table_sendout_key(pd, (uint8_t*)key);
11      uint8_t* value = getExactEntry(tables[TABLE_sendout], (uint8_t*)key);
12      struct sendout_action* res = (struct sendout_action*)value;
13      if(res != NULL) {
14      index = *(int*)(value+sizeof(struct sendout_action));
15          switch (res->actionId) {
16              case rewrite_src_mac:
17                  rewrite_src_mac(pkt, tables, res->rewrite_src_mac_params);
18                  break;
19              case on_miss:
20                  no_op(pkt);
21                  break;
22          }
23      }
24      if(res != NULL) {
25          switch (res->actionId) {
26              case rewrite_src_mac:
27                  return table_forward(pkt);
28                  break;
29          }
30      } else {
31          debug("Packet Drop\n");
32          return;
33      }
34  }
```

Listing 3.19 – Auto-generated Table Code

3.6 Features of Architecture

The modular architecture and clear logical separation among the modules have given us a lot of advantages to incorporate well-sought aspects of flexibility like *Programmability*, and *Portability* into MACSAD as depicted in this section. Besides, the support of a remote controller, a strong feature required for every SDN devices in the current landscape, is explored here.

3.6.1 Programmability

Programmability of a switch can have various aspects including packet processing, switch configuration and management, switch monitoring, or flexible interface towards the remote Controller. Our discussion focuses on Protocol Independence features of a switch to explore programmability nature of it. Protocol Independence is a forte that is achieved by being able to (re-)configure data plane using a high-level language to introduce custom protocol by supporting non-standard protocol header format. MACSAD prefers

P4 language to program data plane applications as it inherently cultivates the protocol independence nature. With P4 our focus remains on the application requirements instead of exploring the protocols used or the target platform.

We understand that the Parser and the Datapath specifying all the pipelines a packet can be a part of in the switch are where programmability can have a significant influence. For every network packet, MACSAD is required to extract headers, assemble the lookup keys from extracted header fields, perform table lookups using the keys, and finally execute the associated actions on the packet. The Parser block in conventional switches require the knowledge of protocol header format to construct the lookup keys by specifying the target header fields (e.g., Ethernet MAC_{source} Address). In contrary, MACSAD posses no priori knowledge of the protocols and protocol header formats. The header format is defined at compile time, and the programmable parser is able to identify the custom protocol headers with the header length and header field width details from P4 program itself. This protocol oblivious parsing is referred to as "Protocol Independent Parser" and explored briefly in 3.6.1.1.

Following up, the Datapath block is required to assemble the lookup key and perform 'match+action' operation before sending out any network packet. Conventional switch achieves this with the knowledge of the exact specificities of the header fields of protocols constituting the lookup key. For MACSAD previous knowledge of protocol details are not necessary. It defines the key by one or more header fields identified internally by offset, length tuples where offset denote where the field starts in the header, and length denotes the number of bits to be included in the key starting from the offset position. With the compile-time discovery of key formats and table lookup implementation MACSAD brings protocol independence to the Datapath block and is explored in more details in 3.6.1.2 as 'Protocol Independent Dataplane'.

Programmability feature is inculcated into MACSAD by the auto-generated code explained in section 3.5. The *Packet Parsing Logic* and the *Control Logic* auto-generated by the *Transpiler* are responsible to bring Protocol Independence (PI) to MACSAD by means of a *Protocol Independent Parser* and a *Protocol Independent Dataplane*, respectively.

3.6.1.1 Protocol Independent Parser

According to P4 abstract model (see Fig. 4), the Parser functionality can be interpreted as post-pipeline editing. P4 parses every packet into a 'Parsed Representation' of it. Every packet header update operations are done over the Parsed Representation, and later the deparser module (Fig. 4) puts the updated Parsed Representation back to the packet serializing the headers in sequence before transmitting it out. However, MACSAD implements inline editing of headers instead of post-pipeline editing as in P4. MACSAD

parses each packet and stores the pointers to the required headers and header fields to facilitate the in-place read-write of the header fields. Unlike hardware switches with deterministic delay from parser module, software switches suffers from higher memory latency resulting in higher delay which increases with parser complexity too. Hence, inline editing of parser is chosen to circumvent the additional deparser module and improve MACSAD parser performance. In MACSAD, header structures are identified from the P4 program and the *Transpiler* module auto generates *enums* for header fields, their offset and bit-masks also known as 'Packet Parsing Logic' part of the 'Datapath Logic Code'. MACSAD Protocol Independence (PI) with Packet Parsing Logic to support custom protocol headers defined with P4 syntax for complex header structures are demonstrated through the Data-center Gateway, and Broadband Network Gateway use cases later in Sec. 4.2. To increase performance and streamline the packet processing pipeline, we circumvent the deparser module by opting for inline editing of packet headers.

3.6.1.2 Protocol Independent Dataplane

The foundation of the Protocol Independent Dataplane is a generic forwarding architecture and a concise set of protocol independent primitive actions which can describe any data plane application. $P4_{14}$ presents a set of primitive actions sufficient to describe any data plane application. By being able to define custom protocols over a target device, P4 achieves protocol independent dataplane. In MACSAD, we implement these primitive actions over ODP APIs as part of Backend APIs, while mapping the P4 program blocks to the PISA forwarding architecture. Protocol Independent Dataplane is the result of these two steps in place in MACSAD architecture. *Transpiler* presents auto-generated code for P4 program over PISA exploring the parser logic and control logic composed of Parser, match-action P4 abstractions to define all the pipelines a network packet will eventually take in the data plane. Backend APIs Table 7 maintain cohesion among parser logic and control logic and help bind them together with its implementation of the primitive functions. The auto-generated header data structures are extensively referenced in Control Logic with no priori knowledge of the protocol itself by MACSAD. We showcase that it is possible to have Protocol Independent Dataplane even with complex header structures, and encapsulation & decapsulation of headers defining complicated pipeline in the Data-center Gateway and Broadband Network Gateway use cases later in Sec. 4.2.

3.6.2 Portability

The absence of a standard programming language and multi-architecture compiler system limits the portability of data plane application. A high-level language is meant to be platform independent and normally offers support for more generic switch pipeline architectures explained in subsection 2.1.2. P4 high-level language too is a platform-

agnostic language leaving the heavy lift of packet forwarding details to the target-specific backend compiler. Moreover, P4 relates to different targets with PISA support as P4 abstractions can be mapped well to the switch architecture. MACSAD blends P4 abstractions and primitives with the Backend APIs in Table 7 towards portable data plane applications. A datapath implementation in software typically consists of two functional realms: (1) Packet handling consisting of the Parser, Table (Match+Action) lookup, and Packet header updater; and (2) Switch resource management functions including CPU, Queue, Memory, Thread, Table, among others. While the first set of functions are mostly auto-generated by the Transpiler in a protocol-independent manner for MACSAD, the second set is target dependent. By creating a set of target-independent implementations of these target dependent switch system libraries/APIs on top of ODP APIs, MACSAD delivers target-independent system APIs turning the code seamlessly portable with a highly-reduced effort across network platforms. This solution can be realized by a simple recompilation of the source code without necessarily sacrificing performance across target platforms. Furthermore, MACSAD brings support to hardware accelerated modules, and other nuances in hardware resource provisioning behind this target-independent system APIs as part of Backend APIs. In its current state MACSAD portability support is limited to forwarding architectures based on PISA. However, by adding support for $P4_{16}$, we are looking forward to bring support for Portable Switch Architecture (PSA). As PSA promise to support different switch pipeline architectures including PISA, in time MACSAD will also be able to bring support for these architectures too.

3.6.3 Contoller Support

Although P4 is only capable of specifying the data plane, it implicitly elaborates the interface between the data plane and the control plane. The control plane manages the P4 tables at runtime. *Transpiler* auto generates the necessary table management APIs to allow the external controller to update the pipeline. MACSAD uses a very expressive naming convention for the table APIs following the general consensus prevailing in various other open source project (OPENDAYLIGHT, 2018). The name of an API contains the table name and table action to explicitly express the function of the API. Considering these details are available in the P4 program itself, the controller can identify the APIs supported by the data plane from the P4 program without any additional input required. This facilitates the developers to bring the support of different controllers over P4 target device easily. The Table 11 shows the auto-generated APIs for the P4 table **sendout** for our reference. Currently, MACSAD provides remote controller support over TCP connection. We provide our own take on SDN controller and offer a simple controller which can create a session with MacS and update the table flow entries of the MacS pipeline at runtime.

Table 11 – Auto-generated Table APIs for Control Plane

Table Name	**Table APIs**	**Remarks**
sendout	sendout_set_default_action	Set the default action.
	sendout_add_table_entry	Add or update an entry.
	sendout_del_table_entry	Remove an entry.

3.7 Concluding Remarks

Introduced in subsection 1.3.1, MACSAD is detailed in this chapter explaining the design & implementation, and its various features. We started this chapter with how MACSAD is composed of different modules to bring flexibility in design, and then followed up with the description of the compilation process to showcase MACSAD compiler system. This is then followed by more detailed dive in into the individual modules and submodules. IR generation (section 3.3), and Backend APIs (section 3.4) descriptions focused on the support of high-level DSL P4 and low-level multi-target SDKs from ODP respectively. While exploring the impact of P4 and ODP, the process of code auto-generation for the packet processing logic of MACSAD interpreted contending features like Protocol Independence & Target Independence.

This chapter provided a comprehensive overview of the design and implementation of MACSAD compiler system and the flexibility it brings to the switch in terms of programmability and portability. We carry this discussion to include performance and scalability evaluation of MACSAD in the next chapter bringing some related works into the mix too.

4 Experimental Evaluation

With this chapter, we now turn our attention to the practical aspects of MACSAD implementation, and evaluate them in line with *Programmability, Portability, Performance and Scalability (3PS)* for varied use cases with different complexities establishing our effort to achieve flexibility in forwarding devices. Common believe dictates that performance comes at the cost of programmability and vice-versa. Hence it is essential to have this evaluation to showcase that MACSAD achieves performance without sacrificing programmability or portability contrary to common belief.

Programmability aspect is attained with the demonstration of support for different use cases, namely, Layer-2 Forwarding (L2FWD), Layer-3 Forwarding with IPv4 (L3FWDv4) and IPv6 (L3FWDv6), Network Address Translation (NAT), Data Center Gateway (DCG), and Broadband Network Gateway (BNG) defined with $P4_{14}$ and $P4_{16}$. We identified these use cases to showcase different features of MACSAD and present varied pipeline complexity to satisfy programmability. Complexity is understood in terms of increasing the number of tables, table entries count (from 100 to 100K), and support of tunneling protocol; i.e., increasing per packet processing time. Evaluation of these diverse use cases can demonstrate MACSAD capability to support for the majority of P4 abstractions, and MACSAD ability to bring the data plane applications to the target platform.

Besides, *portability* necessitates the presence of a data plane application over multiple switch targets. MACSAD brings the use cases written in P4 over to various platforms by compiling the P4 program onto PISA architecture, and in turn supporting all the underlying platforms. We showcase MACSAD use cases running over different platforms spanning ARM, x86, ThunderX, and Octeon. This is achieved and demonstrated over devices like Raspberry Pi2, General Purpose Servers, Virtual Machines, Docker Containers, and Cavium bare metal switches. We also bring the different packet I/O drivers, namely, DPDK, Netmap, Socket_mmap, and vfio_pci into the mix increasing the breadth of our approach to *portability*. Hence, we explore the evaluation of use cases in various configurations over a number of platforms in the following sections.

Moving forward with *3PS* we express *performance* and *scalability* in detail too. Performance and scalability evaluation become apparent due to the nature of testing & evaluation of MACSAD done here for this thesis. We explore different combinations of use cases, platforms and Packet I/Os to evaluate the performance of MACSAD, and withal explore the scalability for different workloads (packet traces, table entries, packet sizes) with different configuration options (e.g., CPU cores). This diversity is extended by the use of different network interface types (Intel, Mellanox) with varied throughput (including

10G, 40G, and 100G). We also explore the increasing number of table entries, and network traffic flows (from 100 to 100K) to explore scalability aspect of MACSAD. Performance evaluation is carried out mostly in Packet Rate/Throughput terms for different use cases while Latency feature too is explored briefly in the process for the use cases. The Packet Rate/Throughput evaluation for the switch data plane adheres to the methods defined in RFC 2544 (BRADNER; MCQUAID, 1999). *Performance* evaluation is the primary feature in our evaluation of MACSAD and is described in detail including the trade-off present due to *Programmability, Portability*, and *Scalability*.

We begin with the description of different testbeds used during our evaluation of MACSAD. We present and analyze the results, and discuss the observed trade-offs and scalability patterns for different workloads and configuration options for the use cases executed on our testbeds. Then, we demonstrate how MACSAD fares in terms of packet rate and scalability against other related works such as T4P4S and OvS. In each experiment, MACSAD is only required to recompile the corresponding P4 source code for any change in the target platform. We execute MacS along with a simple in-house Controller to populate the tables for each use case. The main aim of our measurements is to identify how our proposed MACSAD performs under different configurations, and over different target platforms. We have also described our novel technique on dynamic CPU core (de-)allocation towards a scalable data plane capable of adapting to the workload and system needs in the follow-up chapter 4.5.

4.1 Testbed Details

MACSAD analysis is carried out on multiple testbeds each differing in terms of target platforms, the number of CPU cores, throughput capability of network interface cards, or specific traffic generator used. Each testbed consists of a DUT running MacS and Tester accommodating the traffic generator. The Tester and DUT are connected back-to-back as per the RFC 2544. We use Network Function Performance Analyzer (NFPA) (CSIKOR *et al.*, 2015) and OSNT (ANTICHI *et al.*, 2014) as traffic generators to calculate throughput and latency numbers, and to explore the impact of traffic generator on MacS. Similarly, the DUTs are of types like general purpose server and Cavium bare metal switches. The different testbeds provide different combinations of the Tester and DUT selecting one from each category. We also explore the impact of packet size, burst size and packet I/Os over throughput and latency for different testbeds across use cases. MacS pipeline tables are configured in ways such that it ends up in receiving packets from one port and forwarding them via the other port towards the tester, which in turn analyzes the packet throughput in terms of packets per second (pps) and bits per second (bps), and latency in microseconds (μs). In the following paragraphs, we briefly describe the testbed configurations.

Testbed A

Under this testbed, both DUT and Tester runs on similar configuration, Lenovo ThinkServer RD640 servers with Intel Xeon E5-2620 v2 processors (having 6 cores per socket with 2 threads per core running at 2.40GHz, 1 Numa node) and 64GB of memory running Ubuntu Linux 16.04 LTS with kernel 4.4; each server is equipped with a dual-port Intel X540-AT2 NIC (10G). One of the servers is configured to be the Tester running NFPA (CSIKOR *et al.*, 2015) with a stable version of DPDK (v17.08) and PktGen (v3.4.5) where NFPA test system internally uses PktGen tool with DPDK to continuously replay the test traffic available as PCAP files. Furthermore, the DUT supports multiple Packet I/Os, namely DPDK (v17.08), Netmap (v11.2) and the basic Socket_mmap provided by the Linux kernel.

Testbed B

Similar to Testbed A this testbed also has both DUT and Tester running on similar configuration with Intel Xeon CPU E5-2680 v4 (having 14 cores per socket with 2 threads per core running at 2.40GHz, 2 Numa nodes) and 64GB of memory running Ubuntu Linux 16.04 LTS with kernel 4.4; each server is equipped with a dual-port Mellanox MT27700 Family [ConnectX-4] NIC (100G). The Tester and the DUT have NFPA, DPDK, PktGen, ODP, etc., configured in a similar fashion to Testbed A.

Testbed C

This testbed demonstrates MACSAD over Cavium switches using ODP SDKs. Here, the DUT is a Cavium development board with Octeon TX 83XX chipset (24 CPUs, 64-bit, 1 thread per core running at 1.8GHz, 2 Numa nodes) and 16GB of memory running the specially tuned version of Ubuntu 16.04.5 LTS with Kernel 4.9. It has a single socket for CPUs and two levels of cache (L1d:32K, L1i:78K, L2:8192K). It provides DDR4 controllers with ECC and PCI-Express Gen3 for better performance. The ODP SDKs from Cavium for this board is based on v1.11.0.0 (Monarch), a much older ODP version compared to the current version of v1.19.0 (Tigermoth). This required a number of changes to our MACSAD in order to compile and execute MacS. On the other hand, the tester node has Intel Xeon CPU E5-2680 v4 (having 14 cores per socket with 2 threads per core running at 2.40GHz, 2 Numa nodes) and 64GB of memory running Ubuntu Linux 16.04 LTS with kernel 4.4. We have equipped the Tester with a dual-port Intel XL710 NIC QSFP+ (40G) network connection for experimentation.

Testbed D

Testbed D is again based on Cavium chipset but of a different family. The Cavium based DUT is the R150-T62 server with ThunderX 88XX chipset (48 CPUs per socket, 64-bit CPU op-mode, 1 thread per core running at 2GHz, 2 Numa nodes, two levels of cache with L1d:32K, L1i:78K, L2(shared):16MB) and 64GB of memory running Ubuntu Linux

18.04.1 LTS with Kernel version 4.15. The tester node has Intel Xeon CPU E5-1660 v4 having 6 cores per socket with 2 threads per core running at 2.40GHz, 2 Numa nodes and 64GB of memory running Ubuntu Linux 16.04 LTS with kernel v4.4. We have equipped the tester with a dual-port Intel X540-AT2 NIC (10G) SFP+ Network Connection for experimentation.

Testbed E

We explore a hardware-based traffic generator with this testbed by using OSNT traffic generator and analyzer with MacS. Here, the DUT is a general purpose server with Intel Xeon CPU D-1518 (having 4 cores with 2 threads per core running at 2.40GHz, 1 Numa node) and 16GB of memory running Ubuntu Linux 16.04 LTS with kernel 4.4. It has a single socket for CPU cores and three levels of cache (L1d, L1i:32K, L2:256K, L3:6144K). We have equipped the tester with a dual-port Intel X540-AT2 NIC SFP+ (10G) network connection for experimentation. The tester has Intel Xeon CPU E-5506 (having 4 cores with 2 threads per core running at 2.13GHz, 1 Numa node) and 16GB of memory. The OSNT is running over the NetFPGA SUME board attached to this server. The SUME board is equipped with 4 SFP+ (10G) NICs for the experimentation.

With our 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. Traffic traces have different number (from 100 to 1M) of unique flows randomly generated per use case, but consistent across different packet sizes, limiting the impact of the lookup process and underlying caching system which would depend on the traffic pattern. Our own tool BB-Gen (RODRIGUEZ *et al.*, 2018) is used to generate all the traffic traces used in our experiments explored in detail in section 7.1. In most of the cases, we evaluated different packet I/O drivers for which, when it is not stated otherwise, we used blue circle patterns for DPDK, solid green bars for Netmap and orange dotted patterns for the kernel provided Socket_mmap. All measurements are conducted for 60 sec (BRADNER; MCQUAID, 1999), and every data point in our performance measurements is an average value. Confidence intervals are not used as the results are stable and reproducible for all frameworks while evaluating packet rate of MacS. Latency results are expressed using boxplot though to include average, mean and highest values while showing the outliers too. As the calculated latency values are small, the variance is noticeable and important to consider for analysis. We summarize all the testbeds used for MacS evaluation and analysis in the Table 12.

Table 12 – Testbed Summary

Testbed Name	DUT Details					Tester Details
	Manufacturer	Chipset	Architecture	NIC Name	Maximum Throughput	
A	Intel	Xeon E5-2620 v2	x86_64	Intel X540-AT2	10G	NFPA (s/w based)
B	Intel	Xeon E5-2680 v4	x86_64	Mellanox MT27700 [ConnectX-4]	100G	NFPA (s/w based)
C	Cavium	Octeon TX 83XX	AARCH64	Intel XL710	40G	NFPA (s/w based)
D	Cavium	ThunderX 88XX	AARCH64	Intel X540-AT2	10G	NFPA (s/w based)
E	Intel	Xeon D-1518	x86_64	Intel X540-AT2	10G	OSNT (h/w based)

4.2 Use Case Descriptions

4.2.1 Port Forwarding (PortFWD)

Port Forwarding (PortFWD) is a simple use case where MacS receives network packets from one interface and sends out via the other interface without performing any header update operations on the packet itself. With this, we demonstrate the raw performance of the testbed and help to evaluate the other use cases against a reference value henceforth.

4.2.2 Layer-2 Forwarding (L2FWD)

We demonstrate a Layer-2 switching and forwarding program with MAC address learning feature implemented with MACSAD and a specialized external controller. We implement L2FWD with two separate lookup tables, the first matching on source MAC address and the second on destination MAC address. MacS, following the P4 guidelines, generates controller digests for unknown MAC_{source} address and the arrival port ID. In turn, the controller responds to the digest message by directing MacS to add the MAC_{source} address and arrival Port ID to the corresponding tables. P4 "Exact Lookup" method is used for MAC address lookup in the table and is implemented using ODP based Cuckoo Hash helper library in MacS.

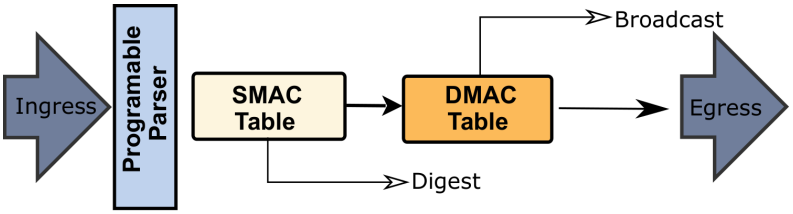


Figure 11 – L2FWD Use Case Pipeline.

Figure 11 shows the basic MacS pipeline which consists of SMAC and DMAC tables. The grey arrows above and below the tables show the action functions executed over the packets in case a table lookup fails. The black arrow shows the sequence of tables a packet traverse in the case of table lookup success. In the event of lookup fail at the first

table SMAC, a packet digest is created at MacS and sent to the controller to perform MAC learning, else no operation is done on the network packet, and the packet moves to the next table in the pipeline. A lookup success at the second table DMAC results in a successful forwarding of the network packet via appropriate output port, whereas lookup fail results in a broadcast of the network packet. The action functions for the two tables are shown at Listing 4.1 and Listing 4.2. Appendix A shows the P4 program for the use case and the dependency graphs for parser and tables.

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

Listing 4.1 – *SMAC*

```
action forward(port) {
    modify_field(standard_metadata
    .egress_port, port);
}
action bcast() {
    modify_field(standard_metadata
    .egress_port, 100);
}
```

Listing 4.2 – *DMAC*

Figure 12 shows the throughput results of MacS running L2FWD use case on our *Testbed A*. The figure shows the throughput results for 3 different packet I/Os across different packet sizes in increasing order while using a single CPU core. As per the prevailing consensus, socket-mmap displays the worst performance whereas DPDK shows better results than both socket-mmap and netmap. While DPDK easily reaches line rate with 256 Bytes packet size, the socket-mmap packet I/O can not saturate the 10G NIC even with 1518 Bytes packet size. Netmap results are somewhere between socket-mmap and DPDK. We observed that Netmap saturates 10G NIC with 1024 Bytes packet size and reaches more than 90% of line rate with 512 Bytes packet size. The line rate for 256 Bytes and 1024 Bytes packet sizes are shown as red dashed and solid line respectively in the Figure 12 for reference.

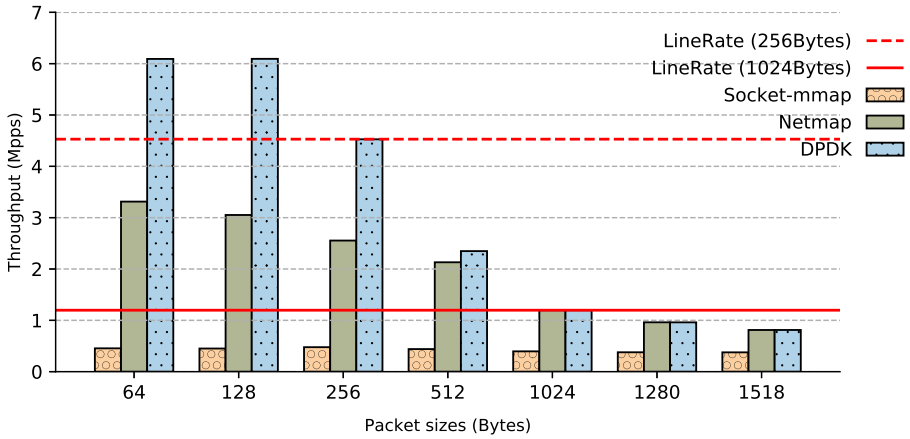


Figure 12 – L2FWD Performance Evaluation (1 core, 100 Table entries) on Testbed A.

4.2.3 Layer-3 Forwarding (L3FWDv4/v6)

With L3FWD use case, we demonstrate Layer 3 IP based forwarding of network packets using either IPv4 or IPv6 network protocol. These L3FWD use cases are implemented with ODP's built-in Helper library for Longest Prefix Match (LPM) based lookup mechanism. ODP provides LPM lookup with 32-bit keys supporting IPv4 forwarding suitable for the L3FWDv4 use case. However, ODP's built-in helper library lacks the support for IPv6 based LPM lookup algorithm. We bring the missing IPv6 forwarding support to ODP by extending the existing LPM library to support 128-bit keys. We implement both Layer-3 Forwarding IPv4 (L3FWDv4) and Layer-3 Forwarding IPv6 (L3FWDv6) use cases in MACSAD using the extended ODP helper library. We keep the structure of tables similar for both use cases for a seamless comparison among them.

The P4 pipeline is implemented using two lookup tables in sequence as done in L2FWD, presented in Figure 13. At first, IP_{dest} Address based lookup is performed at the first table *ipv(4/6)_fib_lpm* along with corresponding actions for standard L3 packet processing (e.g., MAC_{dest} re-writing, TTL/Hop Limit decrement, Output Port selection). This is followed by a matching on the output port in the second table *sendout* and the MAC_{source} re-writing action function. The default action for each table is defined as *Drop* and shown as an additional gray arrow above the table blocks in Figure 13.

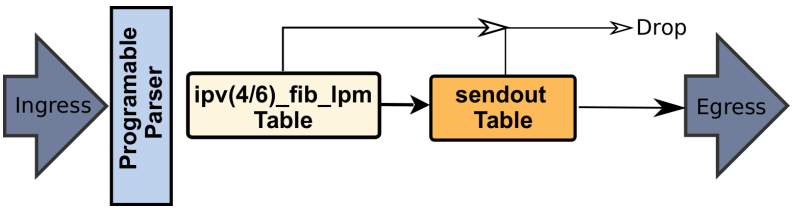


Figure 13 – L3FWDv(4/6) Use Case Pipeline.

Appendix B shows the P4 program for the use cases and the dependency graphs for parser and tables. In the event of a table match with the Destination IP Address (IP_{dest}) as the lookup key at the first table *ipv(4/6)_fib_lpm*, the action function *fib_hit_nexthop* is executed. As shown in 4.3, we update the headers and metadata fields according to the action function *fib_hit_nexthop*. Following up, a table match for the second table *sendout* references the action function *rewrite_src_mac* Listing 4.4 which sets the proper MAC_{source} before the packet is being forwarded via the output port.

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

Listing 4.3 – *ipv(4/6)_fib_lpm*

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

Listing 4.4 – *sendout*

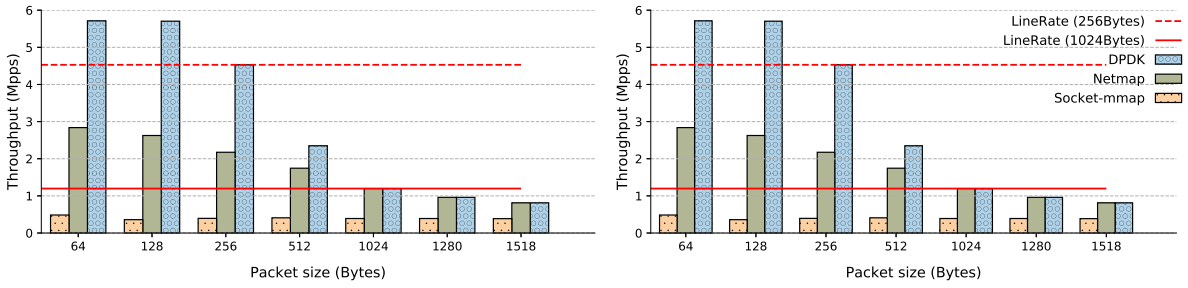
L3FWDv4.

The IPv4 LPM implementation in MACSAD uses a binary tree based lookup algorithm with three tree levels (16-8-8) to achieve the balance between memory consumption and lookup speed bounded at 3 memory accesses per lookup. For this L3FWDv4 use case, we have chosen to go with a 16-bit netmask for LPM lookup resulting in single memory access for each lookup.

L3FWDv6.

The original ODP Helper library lacks IPv6 lookup support. Hence we developed the algorithm for ODP, and implemented table related structures and functions in MACSAD. This LPM lookup algorithm and table implementation are similar to that of DPDK[1] with 15 levels of tables (16-bit 1^{st} level followed by 14 levels of 8-bit each). This is implemented as an extension to the LPM based IPv4 lookup algorithm of ODP where the root node size and the number of table levels are of different values to support IPv6.

Figure 14 shows the throughput results of MacS running L3FWDv4 and L3FWDv6 use cases on our *Testbed A*. The figure shows the throughput results for different configurations of Packet I/Os and packet sizes while using a single CPU core and a table size of 100. The results are similar to L2FWD as MacS pipeline is similarly consists of two tables. Figure 14a and Figure 14b shows the results for L3FWDv4 and L3FWDv6 respectively. We observed that socket-mmap results are the lowest compared to DPDK and NETMAP for both L3FWDv4 and L3FWDv6 use cases. While DPDK performs the best and easily reaches line rate with 256 Bytes packet size, the socket-mmap packet I/O can not saturate the 10G NIC even with 1518 Bytes packet size. Netmap is able to saturate the 10G NIC for 1024 Bytes and more packet sizes.



(a) L3FWDv4 (b) L3FWDv6

Figure 14 – L3FWD (IPv4 & IPv6) Performance Evaluation (1 core, 100 Table entries) on Testbed A.

[1] http://dpdk.org/doc/guides-16.04/prog_guide/lpm6_lib.html

4.2.4 Network Address Translation (NAT)

Towards more complex pipeline, we implemented Network Address Translation (NAT) (EGEVANG; FRANCIS, 1994) use case on MACSAD. NAT remaps an IP address space into another by modifying the IP Headers of network traffic. It enables the hosts inside a private network to mask their identity behind the NAT router. And in turn, it allows the network administrators to implement security measures to protect the private network. With this use case, we explore newer usages of P4 standard metadata and user-defined metadata in the P4 pipeline while adding more number of tables to the pipeline. This use case also explores dynamic table selection depending on the packets processed instead of a fixed sequence of tables explored in the previous L2FWD and L3FWD use cases: table depth of the pipeline differs with the input flow type.

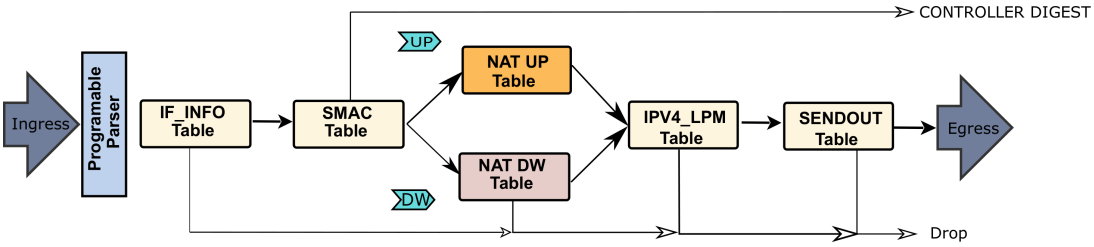


Figure 15 – NAT Use Case Pipeline.

Appendix C shows the P4 program and the dependency graphs for parser and tables for this use case. In our case, the same P4 program satisfies both Uplink (UL) and Downlink (DL) pipeline for NAT. The P4 program has 6 tables in total where each of the Uplink (UL) and Downlink (DL) pipeline consists of 5 tables out of which 4 tables are common among UL and DL as shown in Figure 15. NAT use case pipeline begins with IF_INFO table which identifies whether the network traffic being processed belongs to UL or DL pipeline and update the *routing_metadata.is_int_if* metadata accordingly. After that MAC_{source} address lookup followed by MAC learning when applicable is performed at *SMAC* table. After *SMAC*, the *routing_metadata.is_int_if* value helps to decide the next table and in turn the UL or DL pipeline. If the metadata is set, then NAT_UL, else NAT_DL is referred. In case of NAT_UL table, LPM lookup is performed over IP_{source} Address followed by table action function mapping internal IP_{source} address to external IP_{source} address for every table match. For a table miss, IP_{source} to TCP port mapping is learned instead. Similarly, when *routing_metadata.is_int_if* is not set, a P4 EXACT lookup is performed over TCP destination port ($\text{TCP}_{dstPort}$) as part of NAT_DL table. External to Internal IP address mapping is done for Destination IP Address (IP_{dest}) in the event of a table hit; otherwise, the packet is dropped. Following on, we have two more tables as part of the pipeline: *IPV4_LPM*, and *SENDOUT*. *IPV4_LPM* is the next table in place performing LPM lookup on the IP_{dest} of the network packet. For every successful lookup, it updates the packet with correct MAC_{dest} and sets the Egress port value in the

metadata, and drops the packet in case of a lookup fail. The final table in the pipeline is called *SENDOUT* which perform a lookup upon the Egress port updated in the previous table to set the proper MAC_{source} before the packet is being forwarded. In case of a lookup failure, the packet gets dropped at the table instead of being forwarded. While the use case scenario is explored in Figure 15, the details about the Tables, Actions, etc., are available in the P4 program at Appendix C.

Figure 16 depicts the throughput results of MacS running NAT-UL and NAT-DL use cases on the **Testbed A**. The figure shows the throughput results for different configurations of packet I/Os and packet sizes while using a single CPU core and a table size of 100. Socket-mmap result is the lowest but with constant MPPS (Million packets per second) across different packet sizes. For both UL and DL pipeline, DPDK can saturate the 10G NIC with 512 Bytes or greater packet sizes whereas Netmap achieves that with 1024 Bytes or greater packet sizes only.

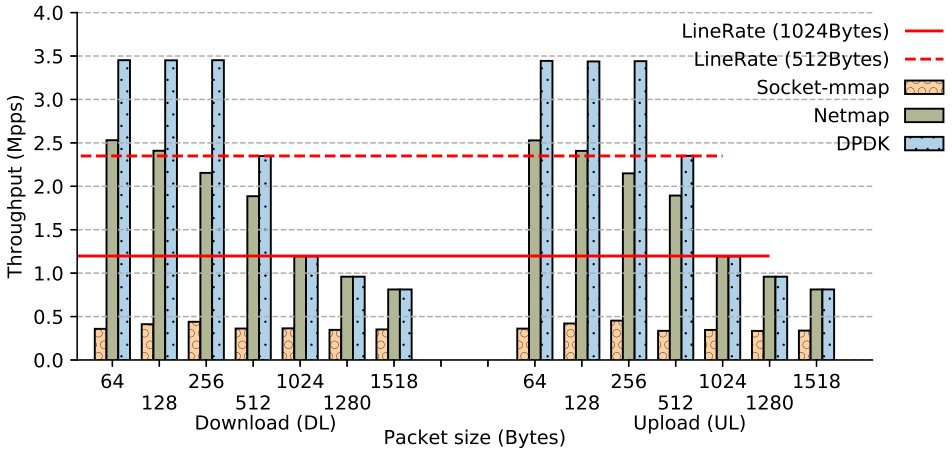


Figure 16 – NAT (UL & DL) Performance Evaluation (1 core, 100 Table entries) on Testbed A.

4.2.5 Data Center Gateway (DCG) with VXLAN

The Data Center Gateway (DCG) use case is the next use case under MACSAD towards a more complex packet pipeline demonstrating new feature support such as tunneling. Tunneling allows transmission of private network data via a public network while being transparent to the routing nodes in the public network. In effect, it allows us to connect multiple logically separated private networks over a public network. DCG use case is developed using Virtual eXtensible Local Area Network (VXLAN) (MAHALINGAM *et al.*, 2014) tunneling protocol as the underlying protocol. VXLAN protocol is an overlay protocol which implements Layer 3 tunnels to connect multiple Layer 2 networks seamlessly. VXLAN requires Virtual Tunnel End Points (VTEPs) at both ends of the tunnel, which can be switches or routers, that (de-)encapsulate the network traffic into a VXLAN header. VXLAN allows creating segments in the network identified by VXLAN Segment

ID/VXLAN Network Identifier (VNI) where communications can only take place within individual segments, not across segments.

The DCG use case scenario using VXLAN is presented in Figure 17. In DCG use case, VXLAN tunnels are used to connect users (Host) with different virtualized web services hosted in redundant servers sharing a common IP address (8.8.8.1 in our example). While the User (HOST) is placed in the public network, the web servers are placed inside a private network as shown in the figure. The VXLAN tunnel exists as between MacS A - MacS B or MacS A - MacS C MACSAD switches where MacS A is the VTEP at one end of the tunnel, and MacS B & MacS C are the VTEPs at the other end of the tunnel. MacsA is the acting data center gateway here in this use case. The VXLAN protocol provides the encapsulation mechanism between VTEPs to transport L2 frames inside UDP packets forwarding the network packets among the HOST and Web Servers. We refer the packet direction towards Web Server as Download (DL), and packet direction towards HOST as Upload (UL). Both DL and UL are considered as two different pipelines of the DCG use case and explored in detail further in this section. Figure 17 shows the DL and UL direction as dotted and solid arrows at the bottom of the diagram with the arrowhead pointing to the respective packet direction. Similarly, the network packet flow across the tables for both DL and UL pipeline is shown in Figure 18.

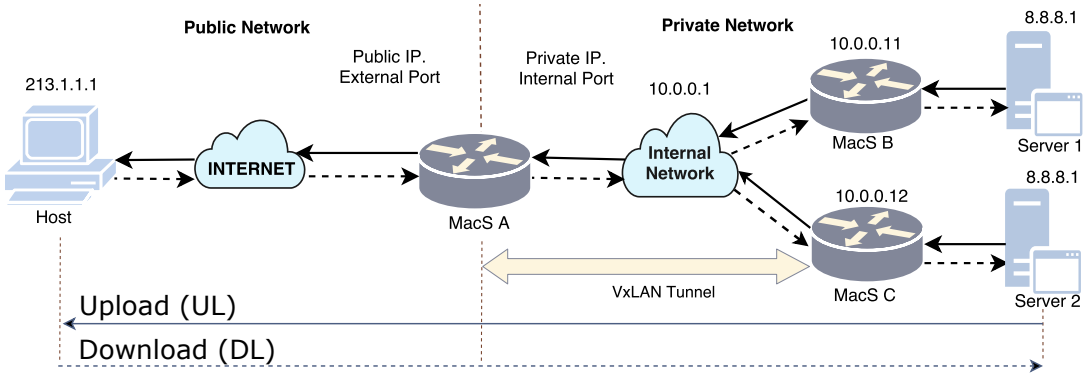


Figure 17 – Data Center Gateway (DCG) use case scenario.

4.2.5.1 Download (DL)

Download (DL) pipeline explains the traffic from the HOST towards Web Server. The traffic is routed via a public network a.k.a internet before reaching the gateway and then traverse through the private network to reach a web service. The User (HOST) & web service bear IP addresses as 213.1.1.1 & 8.8.8.1 respectively. Traffic originated at HOST and routed through the internet to reach the data center gateway MacS A macsad switch which acts as a VTEP too. The network packet enters into a load balancing next hop VTEP decision selecting either MacS B or MacS C, followed by VXLAN header encapsulation. The encapsulation consists of Ethernet, IP, UDP and VXLAN headers. In

the case when MacS B is selected as next hop VTEP, MacS A sets the outer Ethernet header and outer IP header with MAC_{dest} & IP_{dest} of MacS B as shown in Figure 17. In turn, as the last leg of the VXLAN tunnel, MacS B decapsulates the packet and sends it to Server 1.

4.2.5.2 Upload (UL)

The response from web service for the requests from end User (HOST) participates in Upload (UL) pipeline where the network traffic originates at the web service inside a private network behind a VTEP and ends at the HOST situated in a public domain across the internet. To facilitate explanation, we explore the pipeline discussion for the network traffic (response message for HOST) from Server 1. Server 1 response packet reaches the MacS B VTEP at the beginning of the pipeline. As part of encapsulation, MacS B sets the outer Ethernet (MAC_{dest} of A), IP headers, UDP and VXLAN headers in reverse direction towards MacS A VTEP. After that MacS A removes the VXLAN header, rewrites addresses and forwards the packet towards HOST.

Appendix D shows the P4 program and the dependency graphs for parser and tables for this use case. The P4 program satisfies both UL and DL pipelines for DCG. The P4 program has 7 & 8 matching tables for UL and DL respectively resulting in different table depth and pipeline complexity. Both UL and DL pipeline share 6 tables among themselves while other tables are specific to the pipeline itself. Figure 18 illustrates implemented pipelines using different sets of tables as explained here.

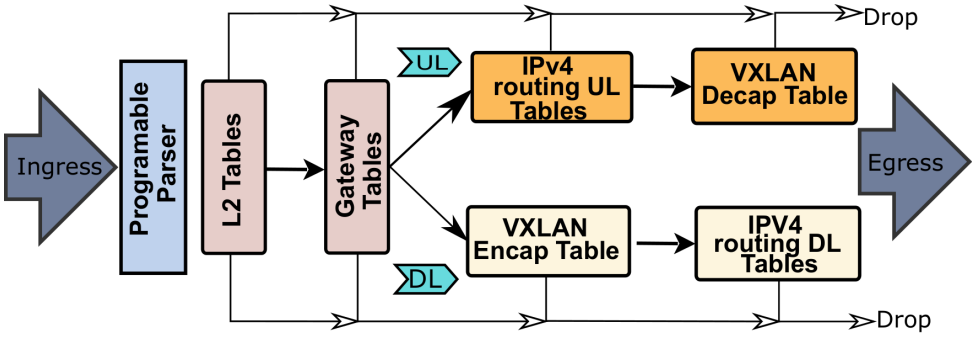


Figure 18 – DCG pipeline featuring the UL and DL table details.

L2 Tables. This is a set of 2 tables allowing DCG to act as an L2 learning switch and to processes ARP packets when necessary. In the case of mac learning, the corresponding tables are updated with new MAC entries appropriately.

Gateway Tables. Since the public network hosts User while the Private network hosts the Servers, this set of 2 tables helps to decide the course of the pipeline to be UL or DL. It also updates the MAC_{dest} for the next hop.

VXLAN Tables. DL pipeline requires encapsulation of VXLAN headers whereas UL performs removal of VXLAN headers. For each pipeline the responsible tables are differ-

ent. Encapsulation and corresponding header field updates are performed using 2 tables in DL pipeline. However, UL requires only a single table to perform decapsulation of VXLAN headers. The encapsulation and decapsulation operations for VXLAN headers are performed using add, remove and copy header functions from the MACSAD *Auxiliary Backend* module.

IPV4 Routing Tables. This set of 2 tables performs the IP based forwarding and is implemented similar to the L3FWD-IPv4 use case explained before. This acts for both UL and DL pipeline in a similar fashion.

Evaluation of DCG use case is done according to the scenario explained in Figure 17. MacS evaluation is carried out at MacS A for both UL and DL pipeline with the help of PCAP files generated by BB-Gen fulfilling all the table requirements. The PCAP traffic files are unique to UL and DL pipelines with different flow details. Separate Table trace files are also created for UL and DL satisfying their pipeline specific set of tables. The PCAP traffic trace includes packets with random host IPs to enable RSS for the multi-core setup and a fixed server destination IP (set to 8.8.8.1). As per our practice, we pre-populated all the tables using the Table Trace file so that every table lookup exits with a match for all the flows in the PCAP based traffic during testing. The load balancing feature is implemented by a checksum function using IP_{source} Address. For DL, MacS adds the right headers and port numbers as per the VXLAN encapsulation. UL pipeline starts with the network packet with VXLAN encapsulation. Hence the smallest packet size for our testing of UL use case is selected as 114 Bytes instead of 64 Bytes to account for the additional 50B overhead of the VXLAN headers. Similarly, the maximum packet size under evaluation is kept to 1280 Bytes instead of 1518 Bytes to accommodate VXLAN headers.

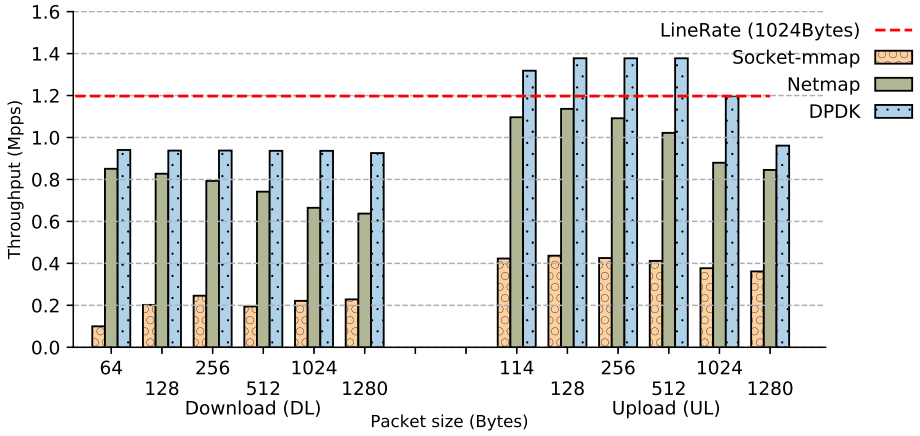


Figure 19 – DCG (UL & DL) Performance Evaluation (1 core, 100 Table entries) on Testbed A.

Figure 19 shows the throughput results of MacS running DCG-UL and DCG-DL pipelines on the *Testbed A*. The figure shows the throughput results for different

configurations of packet I/Os and packet sizes while using a single CPU core and fib size of 100. As expected, socket-mmap behaves poorly and never attain line rate for the 10G NIC in the testbed. However, we observe that for packet sizes greater than 1024B, the MACSAD throughput attains the line rate (10G) for UL pipeline with DPDK packet I/O. This throughput drop from L2FWD and L3FWD use cases is the result of a more complex pipeline with a higher number of tables. We observe a performance difference between UL and DL: UL throughput is higher than DL throughput value. After exploring the P4 code thoroughly and analyzing the packet processing across different tables, we observed that DCG-UL performs VXLAN decapsulation at the end of the pipeline whereas DCG-DL performs encapsulation in the middle of the pipeline. Encapsulation step in DL refreshes cache which is leveraged by tables further down the pipeline. Contrary to it, DCG-UL faces higher impact of cache miss by putting decapsulation step at the end of the pipeline. Our analysis points out that the encapsulation step does not have a more significant impact on throughput than decapsulation step. Hence this result seems counter-intuitive. Further investigation revealed that DCG-DL pipeline has an additional table with EXACT Lookup method compared to the DCG-UL pipeline. We have observed that the impact of an additional table with "match+action" is significantly higher in MACSAD pipeline. As a result, a higher throughput value is measured for DCG-UL compared to DCG-DL whose throughput is penalized by an extra Table in the pipeline. This discloses an important behavior of MACSAD pipeline, and allows us to plan the P4 programs accordingly for more complex use cases.

4.2.6 Broadband Network Gateway (BNG)

BNG, also known as Broadband Remote Access Server (BRAS) (DIETZ *et al.*, 2015), is an integral part of today's Internet and handles the majority of access network traffic implementing network policies and services that an Internet Service Provider (ISP) defines per subscriber. It is also responsible for providing services such as triple play (Internet, Voice, TV) to Customer Premise Equipment (CPE) which represents the triple play communication devices (Telephone, PC, Set-top box) always connected to the network using an access technology (e.g., Digital Subscriber Line). Functions of a BNG also include: Authentication, Authorization and Accounting (AAA) and session management; Packet encapsulation/decapsulation; ARP proxy; NAT; QoS enforcement etc. Similar to our DCG use case, BNG use case also integrates a tunneling protocol using Generic Routing Encapsulation (GRE) tunneling. GRE (FARINACCI *et al.*, 2000a) tunneling protocol allows encapsulation of different network layer protocols over an IP network.

The BNG use case scenario is presented in Figure 20. It handles traffic between a private network and an external public network. Here the important data plane functions are divided into an Upload (UL) and a Download (DL) pipeline. The UL is referred for

network traffic from private network towards public network whereas the traffic in the reverse direction is referred as DL pipeline. Both UL and DL pipelines are shown clearly in the Figure 20 with dashed and solid arrows respectively. Under this use case, the tunnel is implemented in the local private network where CPE resides; the Server exists across the public network. The private network is represented as an Access Network while the public network is represented as the Internet as in the Figure 20. The use case explains the traffic between the CPE behind the Access Network and an external Server across the Internet.

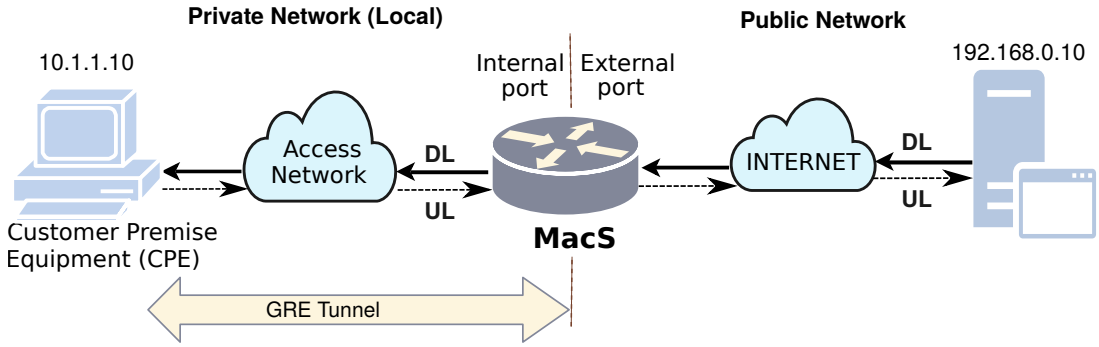


Figure 20 – BNG use case illustrating a subscriber and an external public service.

4.2.6.1 Upload (UL)

UL pipeline explains traffic from the user client a.k.a CPE towards the external Server. A home gateway encapsulates the network traffic packets from the CPE with GRE protocol before forwarding towards the MacS BNG via the Access Network. MacS performs Layer 2 address learning to update the MacS tables and rewrites Layer 2 addresses of the network packet as required. Afterward, MacS verifies the user ID and decapsulates the GRE headers. Followed by, MacS performs NAT over the network packet where NAT specific tables of MacS rewrite the inner headers with the appropriate source IP address and TCP ports. Finally, IPv4 forwarding takes place and the output port is identified to send the packet towards an external server (192.168.0.10).

4.2.6.2 Download (DL)

DL traffic path originates at Server and ends at the user client (CPE) as shown in Figure 20. The server (192.168.0.10) sends TCP traffic over the Internet back to the user client (10.1.1.10) via MacS through the external interface. The MacS performs NAT and updates the packet with the correct destination IPv4 address and TCP port. NAT operations are followed by addition of the point-to-point GRE tunnel header. MacS then updates the IPv4 outer header and verifies the user ID[2]. Then MacS finalizes with the IP

[2] and applies QoS policies: feature not implemented in the current prototype.

packet forwarding by selecting the next hop and output port towards the home gateway which performs GRE decapsulation before sending the network packet to CPE.

This BNG use case is written in $P4_{16}$ to demonstrate MACSAD support for the same. Appendix E shows the P4 program and the dependency graphs for parser and tables for this use case. For both UL and DL, the same P4 program is used where the tables selected and their sequence in the pipeline are decided at runtime. All MACSAD specific discussion and evaluation of BNG use case is done at MacS as shown in Figure 20. The P4 program has in total 9 tables and both UL & DL pipeline has 6 tables each out of which 3 tables are common and 3 tables are specific to the pipeline. For the sake of simplicity, we explain both the pipeline as a sequence of four sets of tables as shown in Figure 21. The table sets are as follows:

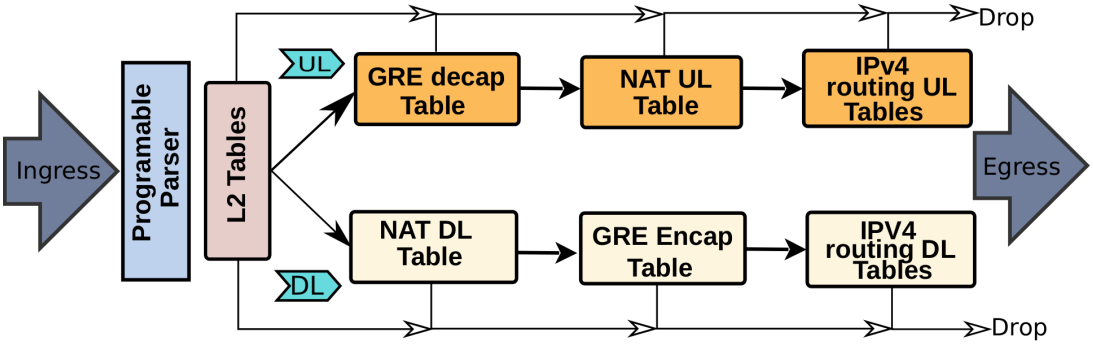


Figure 21 – Implemented BNG pipeline featuring the main UL and DL tables.

L2 Tables. This set of two tables allows MacS to act as an L2 learning switch similar to L2FWD use case and processes ARP packets coming from the user client (CPE). While performing L2 learning, MacS updates the corresponding tables appropriately. L2 learning helps MacS to discover and save the connected devices in the network. Additionally, L2 tables identify and separate the different UL and DL traffic, and configure the packet metadata either as External or Internal.

NAT UL/DL Tables. This set consists of 2 tables; One table each for UL and DL. As the CPE is residing behind a private IP network, NAT is necessary to map internal IPv4 addresses and TCP ports with external address and Port, and vice-versa. For example, MacS updates network packet header and translate IPv4 addresses and TCP ports of network packet headers for UL traffic from CPE towards Server. Packets without corresponding entries in the NAT table result in table miss and eventually dropped in the MacS pipeline. These NAT tables are implemented similarly to the NAT use case described before.

GRE Encap/Decap Tables. With two tables, this set of tables perform the tunneling feature integral to the BNG use case where the GRE tunnel exists between the CPE and the MacS in the private network. For DL traffic, the relevant table encapsulates packets destined to the internal network with a GRE packet header (FARINACCI *et al.*,

2000b) identifying the user to establish a user session. In the reverse direction (UL), for the CPE-originating packets, we perform decapsulation by removing the GRE headers with the help of the relevant table. The SetValid (header add) for encapsulation and SetInvalid (header remove) for decapsulation are implemented in the *Auxiliary Backend* module using ODP APIs explained in section subsection 3.1.2.

IPv4 UL/DL Tables. This set is made of 3 tables and is implemented similarly to the L3FWDv4 use case. The forwarding tables have entries with next hop details such as IP addresses and output port, and take forwarding decision based on table lookup with IP_{dest} address. For every successful table lookup, a number of actions are performed over network packet headers whereas lookup fail results in packet drop: (i) MAC_{dest} and MAC_{src} address update, (ii) Time-to-live (TTL) decrement, and (iii) Output port metadata update.

We test and evaluate MACSAD for the BNG use case at MacS as shown in the Figure 20. Two different types of traffic traces were used: UL path coming from the CPE (IP address 10.1.1.10) to an Internet server (IP: 192.168.0.10), and DL path from the server back to the CPE. Our BB-Gen tool created the traffic in the form of PCAP traffic trace files where the PCAP files can either act as the server or the CPE depending on its flow contents. The traffic traces include distinct flows with randomly generated unique header details suitable for a worst-case scenario. Furthermore, we also created the Table Trace files which have the details for all the tables of MacS. Before every run of the experiment, we pre-populate the tables using the Table Trace files with the help of a remote controller to avoid any table miss condition. UL pipeline starts with the network packet with GRE encapsulation. Hence the smallest packet size for our testing of UL use case is selected as 82 Bytes instead of 64 Bytes to account for the additional 18 Bytes overhead of the GRE headers. Similarly, the maximum packet size under evaluation is kept to 1280 Bytes instead of 1518 Bytes to accommodate GRE headers.

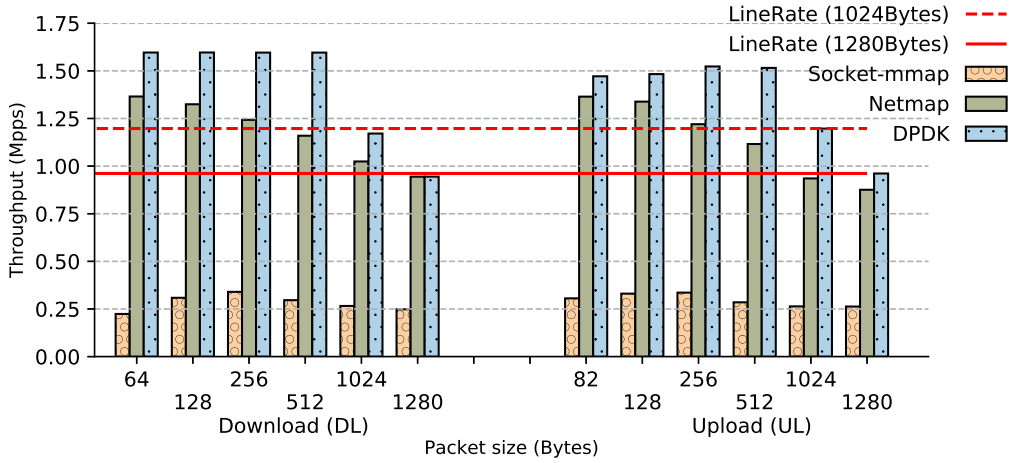


Figure 22 – BNG (UL & DL) Performance Evaluation (1 core, 100 Table entries) on Testbed A.

Figure 22 shows the throughput results of MacS running BNG-UL and BNG-DL use cases on the *Testbed A*. The figure shows the throughput results for different configurations of packet I/Os and packet sizes while using a single CPU core and fib size of 100. We observe that throughput of DPDK Packet I/O is highest followed by Netmap and Socket-mmap respectively. While DPDK saturates the 10G NIC of the testbed with 1024 Bytes packet size or more, Netmap reaches line rate only for 1280 Bytes packet size or more.

We observe near equal performance for both UL and DL pipeline: DL performs slightly better than UL. Analysis of the UL and DL table organization reveals that both have the same number of tables. In the case of UL, decapsulation step is followed by NAT whereas for DL use case NAT comes before encapsulation. After evaluating the underlying MACSAD code for both the pipeline, we observed that UL results in slightly more number of cache-miss than DL, and hence the throughput difference between them. Moreover, due to the higher cache-miss Netmap performs somewhat better compare to DPDK for UL pipeline as DPDK has a higher cache footprint (GALLENMüLLER *et al.*, 2015).

4.3 MacS Evaluation & Analysis

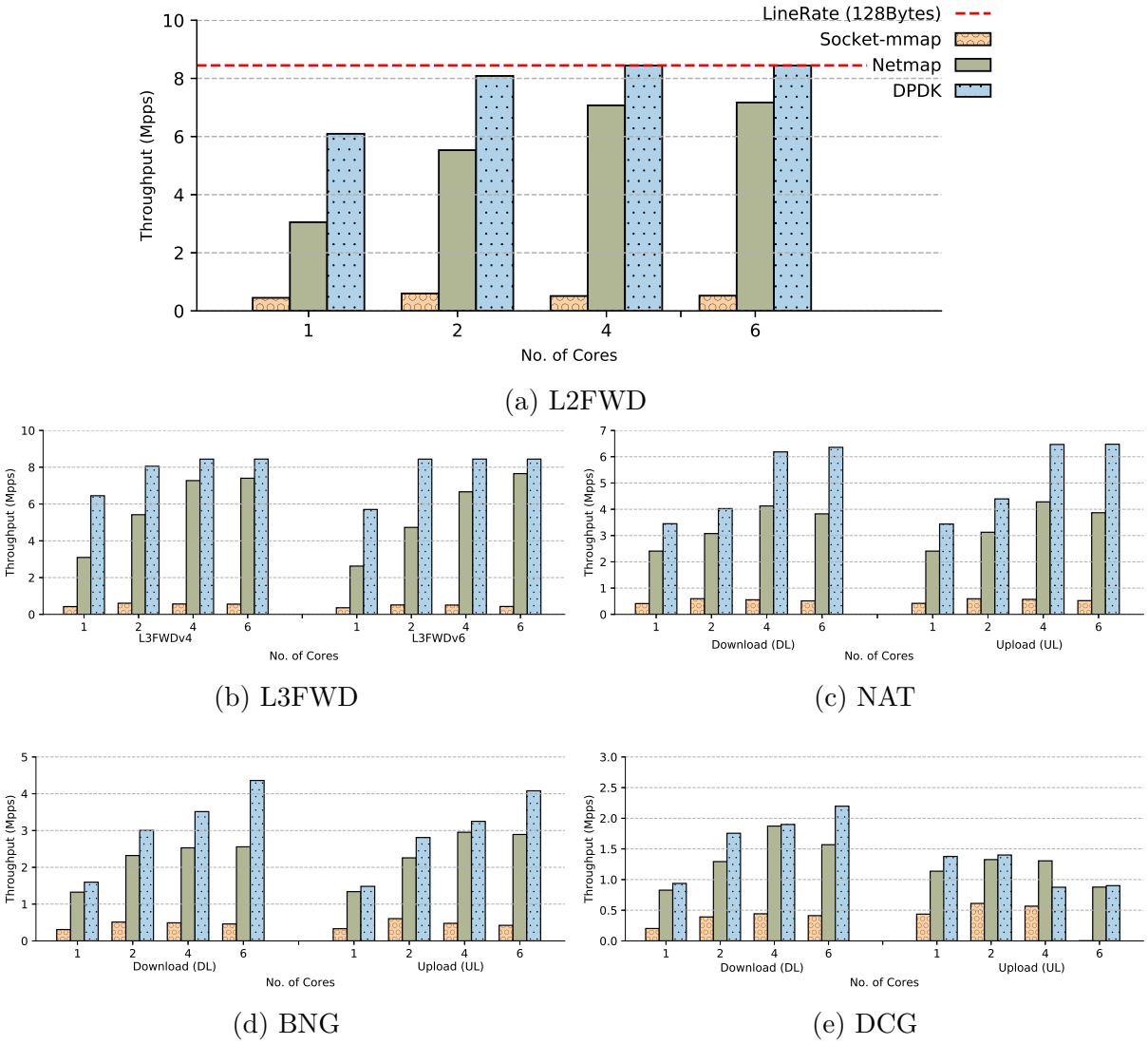
We presented our approach towards MACSAD and its evaluation in section 1.3. We will evaluate MACSAD for *programmability, performance, scalability* & *portability.* Packet Rate (in Mpps) or Throughput (in Gbps), and Latency are the network metrics considered for this activity. In this section, we show how MacS with different use cases shown in section 4.2 performs across different configurations and workloads. Different configurations of MACSAD varies with the variant packet sizes, number of CPU cores, different burst sizes, different Packet I/O drivers, and FIB sizes. More to this diverse set of configurations, we also expand our MACSAD evaluation to different platforms using all the testbeds shown in section 4.1.

4.3.1 Packet Rate Analysis

In this section, we evaluate the MacS packet rate results for different parameters such as number of cores and FIB sizes. We will also explore MacS results over different target platforms too for these defined parameters.

Figure 23 presents the packet rate in Mpps for MacS running with different number of CPU cores, i.e., 1, 2, 4 & 6 cores and 100 Table entries. Results for all our use cases are shown in 5 different sub-figures. Each sub-figure also shows the results of the three supported Packet I/O drivers: Socket-mmap (yellow), Netmap (green), DPDK (blue).

This evaluation is done with 128 Bytes of packet size for all the use cases because the smallest packet size 64 Bytes does not apply to DCG & BNG use case.



(a) L2FWD

(b) L3FWD

(c) NAT

(d) BNG

(e) DCG

Figure 23 – Packet Rate for all Use Cases with different CPU Cores (128 Bytes, 100 Table entries) on Testbed A.

To begin with, we present the result of the L2FWD use case in Figure 23a. MacS can reach line rate with 4 & 6 CPU cores for 128 Bytes packet size while with 2 Cores the packet rate crosses 90% of the line rate. Next, Figure 23b presents the L3FWD use cases where L3FWDv4 is on the left and L3FWDv6 is on the right side of the figure. Both use cases achieved line rate for 4 & 6 CPU cores. In fact, L3FWDv6 achieves line rate with two cores also unlike L2FWD and L3FWDv4 use cases which fell short by a small margin. Figure [23c,23e,23d] demonstrate the packet rate results for the next three use cases i.e., NAT, DCG, and BNG. Figure 23c shows results similar to L2FWD and L3FWD use cases: increasing packet rate with more CPU cores. The maximum packet rate achieved is 6.4 Mpps and 6.5 Mpps for NAT-DL and NAT-UL respectively with DPDK Packet I/O, while

with NETMAP MACS only able to reach for 4.1 Mpps and 4.3 Mpps. Figure 23d shows BNG results with DPDK Packet I/O reaching a maximum of 4.4 Mpps and 4.1 Mpps for BNG-DL and BNG-UL respectively while Netmap & Socket-mmap lagging behind for all CPU Core combinations. Finally, the DCG results shown in Figure 23e appear to be the lowest among all other use cases as the maximum packet rate for DCG-DL and DCG-UL are 2.2 Mpps and 1.4 Mpps respectively. Interestingly DCG-UL shows a lower packet rate with 6 CPU cores compared to 2 & 4 CPU cores. In fact, it is apparent that the packet rate decreases a little from 4 cores to 6 cores in multiple cases or at best remain constant. This decrease in packet rate is mostly attributed to the hyper-threaded CPU core. In hyper-threading, there are two logical cores sharing the same physical core working with the same CPU resources. A CPU intensive application that needs high throughput utilizes more CPU resources and get impacted severely when it associates its threads with the logical cores of the same physical core. With a more complex pipeline, the packet processing becomes more CPU intensive for MACS and in turn the impact of use case complexity over the hyper-threaded logical core is prominent in the figure as packet rate decreases. The other anomaly in the result observed is that DPDK packet rate is affected more than Netmap. This is similar to the results of Figure 22 explained in subsection 4.2.6. In the case of BNG and DCG, we have a higher number of table lookups, and also header addition and removal operations are executed per network packet which are memory intensive operations. These results in higher cache misses and larger impact on the packet rate for DPDK packet I/O due to its higher cache footprint (GALLENMüLLER *et al.*, 2015).

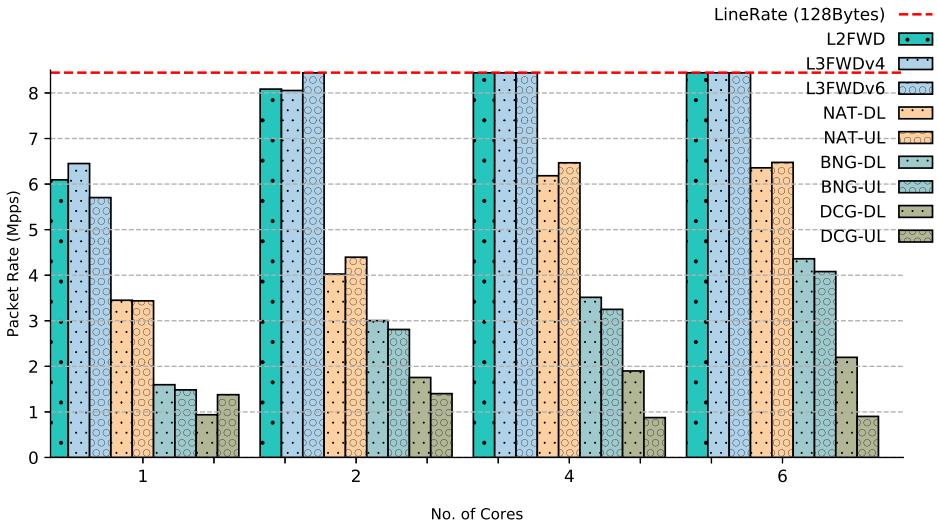


Figure 24 – Packet Rate for Different Use Cases and CPU cores (128 Bytes) on Testbed A.

With Figure 24 we compare the MACS packet rate results for all the use cases with an increasing number of CPU cores using DPDK Packet I/O. Each use case is identified with a different color: L2FWD (teal), L3FWDv4 & L3FWDv6 (sky blue), NAT (yellow),

BNG (cyan), DCG (green). In addition, we use dotted or circle pattern to identify 'IPv4 & Download/Downlink' or 'IPv6 & Upload/Uplink' respectively. The red dotted line in the figure shows the line rate for 128 Bytes packet size chosen for this experiment. It consolidates our observations from section 4.2 and Figure 23 to show how packet rate for different use cases compare against each other.

Next we present MacS performance results over Cavium bare metal switch with ThunderX architecture as part of the *Testbed D*. The experiment is carried out with NICVF packet I/O driver, 4 CPU Cores and 10G SFP NIC for different use cases. Figure 25 shows the packet rate for different use cases and different packet sizes. The red dotted line shows the line rate for 512 Bytes packet size. Similarly, the different use cases are represented in different colors: L2FWD (Gray), L3FWDv4 (Light Purple), L3FWDv6 (Purple), NAT-DL (Light Green), NAT-UL (Green). We observed that MacS saturate the 10G NIC with 512 Bytes or larger packet sizes. L3FWD use cases performed better than the L2FWD use case as we observed in Testbed A. Also, NAT use cases have a lower packet rate compared to L2FWD and L3FWD as expected.

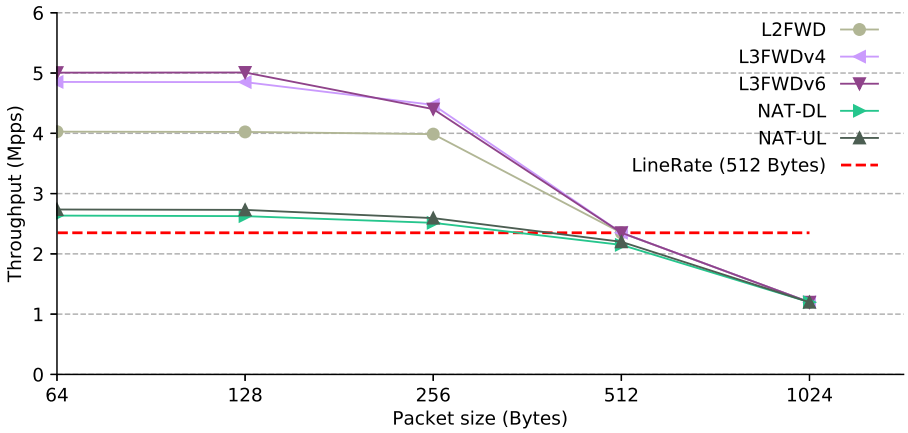


Figure 25 – Packet Rate of different Use Cases & packet sizes. (100 Entries, 4 CPU Cores) on Testbed D

We further explored MacS results over Cavium ThunderX architecture by comparing the packet rate against different number of CPU Cores for all the use cases as shown in Figure 26. The figure shows the PortFWD use case as a solid red line and represents the maximum packet rate with the underlying ODP. This is used here as a reference to the maximum packet rate possible on this testbed (Testbed D). The packet rate for different use case compared against each other similar to our previous observation in Figure 25. Although the packet rate is lower compared to our Testbed A, MacS shows a linear increment with increasing number of CPU cores. This is attributed to the coherent cache system using Cavium Coherent Processor Interconnect™ (CCPI) as part of Cavium ThunderX architecture. With a mere total 16 MB of coherent cache, MacS performance is restricted to a lower packet rate. However, at the same time, fully shared

coherent cache allows MACS to scale linearly across the different CPU cores as the cache miss/hit impact is distributed across cores evenly.

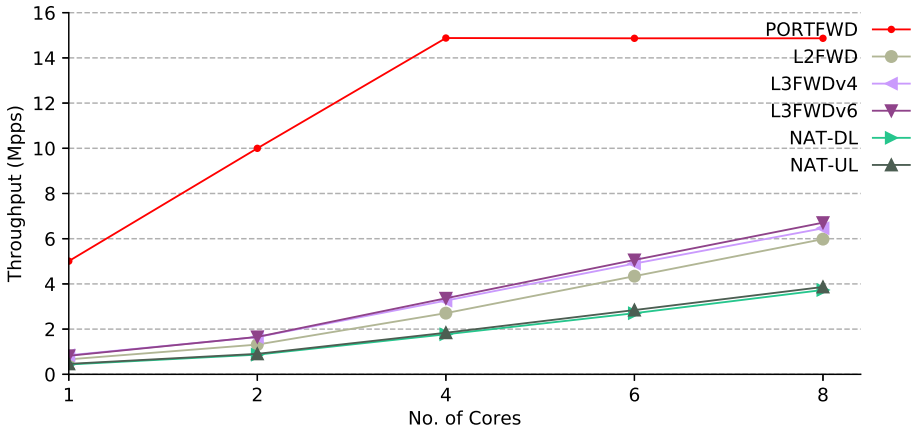


Figure 26 – Packet rate for different Use Cases & CPU Cores. (64 Bytes, 100 Entries) on Testbed D

Following, we analyze various factors influencing the packet rate of MACSAD use cases. For MACS evaluation, we identified a number of these factors: FIB Size, Burst Size, and Traffic Generator.

4.3.1.1 Impact of FIB Sizes

In this section, we evaluate packet rate results for different use cases of the MACS against increasing FIB sizes, i.e., the number of table entries. For this experiment, the packet rate measurements are obtained with **Testbed E**. This experiment is performed over MACS with different packet sizes (128 Bytes & 256 Bytes) and different FIB sizes (100, 1K, 10K, 100K) as shown in Figure 27. MACS configuration consists of 2 CPU cores and DPDK Packet I/O. The x-axis of the Figure 27 shows the results for different use cases in two segments where the left segment is valid for 128 Bytes packet size and the right segment is valid for 256 Bytes packet size. The segments are composed of multiple groups each representing a distinct use case, whereas every group demonstrates multiple bars each representing a unique FIB size. We observed that L2FWD & L3FWD achieve line rate with 256 Bytes or more packet size whereas NAT & BNG achieve line rate with 512 Bytes or more packet size. Hence for the sake of brevity, we show packet rate results for 128 Bytes and 256 Bytes only ignoring the results reaching line rate in the figure. It is obvious that complex use cases are unable to achieve line rate due to system bottlenecks as MACS needs to perform more actions on every network packet. This effect manifolds for smaller packet sizes as the number of packets to be processed increases for the same line rate.

In addition, the Table 13 shows how the packet rate is decreasing between 100 and 100K Fib sizes in percentage and numbers(Mpps). This experiment is able to uncover

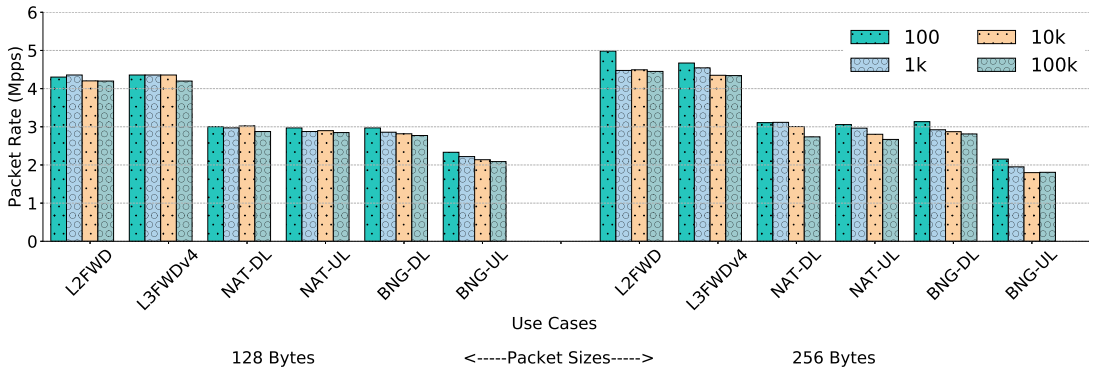


Figure 27 – Packet rate for different Use Cases, FIB sizes. (2 core, DPDK) on Testbed E

Table 13 – Packet Rate Behavior for Different FIB Sizes

Packet Sizes	Use Cases					
	L2FWD	L3FWDv4	NAT-DL	NAT-UL	BNG-DL	BNG-UL
	Percentage Decrease in Packet Rate (100 and 100K FIB size)					
128 Bytes	2.42	3.64	4.22	4.12	6.71	10.49
256 Bytes	10.68	7.05	11.96	12.71	10.24	16.08
	Decrease in Packet Rate in Mpps (100 and 100K FIB size)					
128 Bytes	0.10	0.16	0.13	0.12	0.20	0.24
256 Bytes	0.53	0.33	0.37	0.39	0.32	0.35

MacS behavior towards increasing FIB size across different use cases. Interesting to note that the degree of impact of FIB size appears to be different for different use cases. As the pipeline complexity grows, the percentage decrease in packet rate also goes upwards. To understand the behavior better, we looked into L3FWDv4 and BNG-UL use cases which have 2 and 6 lookup tables shown in Table 16. BNG-UL also differs from L3FWDv4 by implementing 4 header removal and a higher number of header field update actions. With this, we can safely assume that BNG-UL is a more memory intensive pipeline than L3FWDv4. Now with the increase in FIB size, the table lookup time will increase putting pressure on the memory intensive operations in the BNG-UL use case. Although the impact of the delay of per table lookup time is nearly similar for both L3FWDv4 and BNG-UL, BNG-UL degrades more in accordance to three times more number of tables in place as observed for 128 Bytes packet size. For 256 Bytes packet the difference between BNG-UL and L3FWDv4 decrease by small value as the number of packets per second comes down due to the higher packet size, and hence MacS behavior remains as expected.

4.3.1.2 Impact of Burst Sizes

This experiment is focused on understanding how MacS behaves with changing burst size configured for DPDK Packet I/O. In Figure 28, we demonstrate packet rate for

L2FWD & BNG-DL use cases with different packet sizes calculated for a range of burst sizes. This experiment is done with a configuration of 100 Fib Size and DPDK Packet I/O on our Testbed E. The different burst sizes are represented in different colors and pattern combination: 8 (teal, dot), 16 (teal, circle), 32 (blue, dot), 64 (blue, circle), 128 (yellow, dot), 256 (yellow, circle), 512 (cyan, dot). The left side of the figure shows results for L2FWD use case while the right side is for BNG-DL use case. As both the use cases reach line rate at 512 Bytes packet size, we only present here the results from 64 Bytes to 512 Bytes packet sizes.

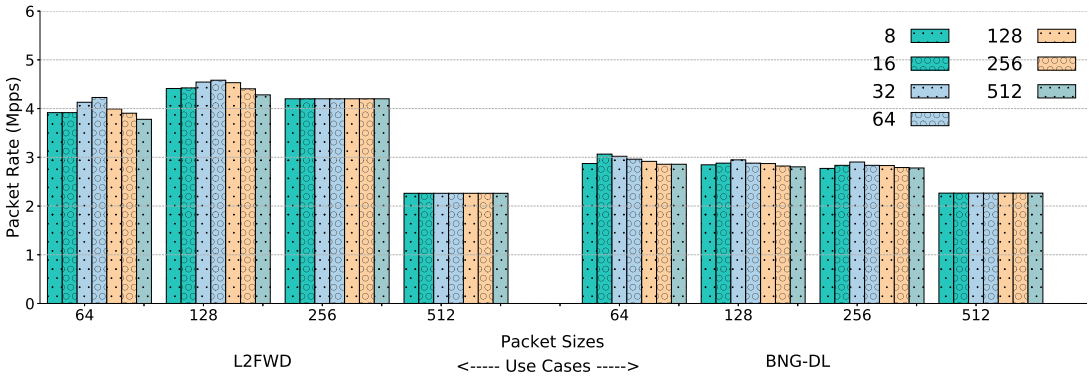


Figure 28 – Packet Rate for different Use Cases, burst sizes. (100 Entries, 2 CPU cores, DPDK) on Testbed E

We tried to identify the best-suited burst size for MACSAD. The current state of the art focuses on the burst size for traffic generator, but there is a lack of material evaluating how software switches behave when set with different burst sizes. Our focus here is to analyze burst size impact on the network traffic receiving end instead for outgoing traffic. Important to note that ODP, DPDK, and Netmap all have default burst size configured as 32 for most of the NIC drivers supported under their umbrella. We tried to validate this unwritten consensus among the research community with MACSAD. The result, Figure 28, shows that the packet rate displays a convex shape for increasing burst size from 8 to 512 achieving maximum at 64 in majority of the configurations. As a result, we identified '64' as the best burst size for MACSAD contrary to the widely accepted burst value of '32'.

4.3.1.3 Impact of Traffic Generators

To analyze the impact of traffic generator on MACSAD, we identified two traffic generators: NFPA (software based using pktgen) and OSNT (hardware based over Net-FPGA SUME). Figure 29 demonstrates packet rate results for MACS in *Testbed E* executing L2FWD, L3FWDv4, NAT-UL, and BNG-UL use cases with 128 Bytes, 256 Bytes, and 512 Bytes packet sizes. The different use cases are represented by a range of colors: L2FWD (cyan), L3FWDv4 (yellow), NAT-UL (blue), BNG-UL (green). Results

of NFPA and OSNT are differentiated by the use of dotted and circular pattern in the graph bars.

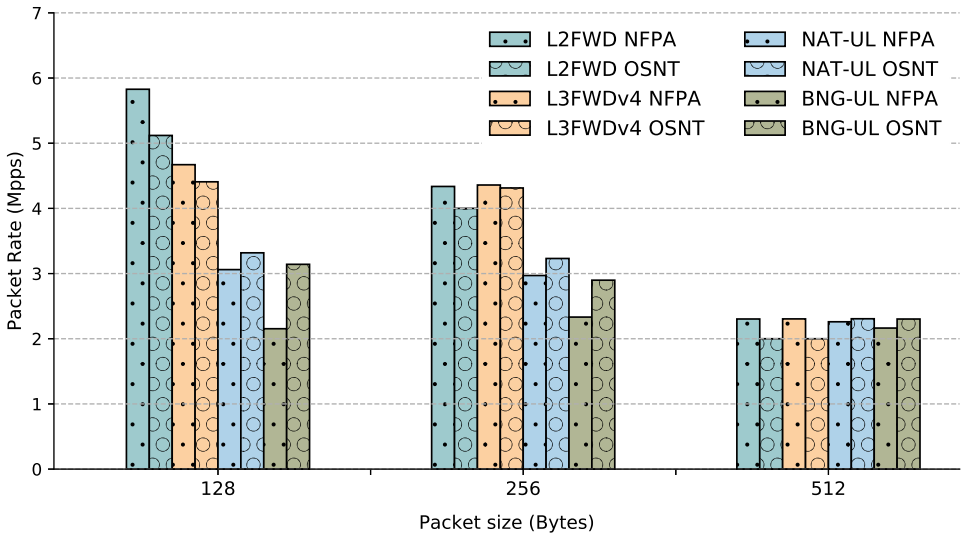


Figure 29 – Forwarding performance of different Use Cases, Pkt sizes, TG. (100 Entries, 2 Cores) on Testbed E

Figure 29 presents two interesting MacS behavior to analyze. It is clear from the figure that the packet rates are different for NFPA and OSNT, but the impact of traffic generator varies according to the MacS configurations. The difference in packet rate between NFPA and OSNT decreases with increasing packet sizes. The BNG-UL results in the figure clearly show this behavior. With an increase in packet size, the number of packets processed by MacS is reduced and so the impact of traffic generator.

Table 14 – Processing Time for a Single Network Packet

Use Cases	Single Packet Processing Time (ms)
	Packet Size (128 Bytes)
L2FWD	0.164103911
L3FWDv4	0.155009921
L3FWDv6	0.175315568
NAT-DL	0.289804672
NAT-UL	0.290892166
BNG-DL	0.626409421
BNG-UL	0.674308833
DCG-DL	1.066552901
DCG-UL	0.725847427

The other interesting MacS behavior from this activity is how the complexity of the use cases also has an effect on this experiment. We observe that L2FWD and L3FWDv4 use cases have a higher packet rate with NFPA whereas NAT-UL and BNG-UL

perform better with OSNT traffic generator. Table 14 shows the time taken to process a packet by MacS for different use cases. By correlating this value with results from Figure 29, we can say that the impact of OSNT traffic generator is more significant when per packet processing time is higher. The main difference between NFPA and OSNT is the way they send out traffic. NFPA pushes out traffic in a burst manner, i.e., it transmits packets in a batch of multiple packets of different sizes such as 32, 64, etc. On the contrary, OSNT is a hardware-based solution and transmits packets with a fixed inter packet gap (IPG), i.e., it waits for a time equal to the IPG value between each packet. Due to this difference, MacS either receives packets in a large batch or one packet at a time which can be understood as a smaller burst size. Hence similar to Figure 28 results, here also we see an increase in packet rate of NFPA. When the single packet processing time is higher, MacS needs to wait for that time before processing the next packet from the batch of the packets received. The NIC drivers will drop the receiving packets if the RX queues of the incoming NIC is full and will start receiving new packets as the packets in RX queues will be processed by MacS. Moreover, it is possible that no new packet arrives at the NIC when MacS is ready to receive new packets in case of NFPA as NFPA transmits packets in batches only. By implementing a lower burst size, or using IPG at the traffic generator, we can reduce the number of instances when incoming packets are dropped due to RX queue full at MacS. Because of this, OSNT performs better for complex use cases as MacS spends less time waiting for the new packets and utilizes the CPU cycles optimally.

4.3.2 Latency Analysis

Latency evaluation experiments are performed in *Testbed E* with OSNT as the traffic generator. We use OSNT on a NetFPGA SUME board for high precision latency calculation. The packets are time stamped in hardware just before transmitting (TX) and just after receiving (RX) at the traffic generator (i.e., OSNT) side. By avoiding any software latency and queuing delays at the SUME board, OSNT can achieve very high resolution up to 6.6ns. The time stamp values are added in the packets at a predefined position and often in the payload of the packets to avoid stressing the parser and packet processing, and avoid any unnecessary increase in delay while calculating latency. By adding a timestamp in the payload, the latency evaluation remains immune to the tunneling protocols too as addition and deletion of headers do not impact the payload of the packet.

OSNT uses configuration scripts internally to direct NetFPGA SUME board on how to time stamp TX and RX packets during the experiment. The scripts specify the position in the packet where to add the timestamp, how many packets per second to time stamp and other necessary configuration parameters for SUME board. The timestamp

header has two fields: ts_rx & ts_tx of 8 Bytes each. The latency measurement methods follow the approaches mentioned in (KAWASHIMA *et al.*, 2017). OSNT transmits packets at 99% of the line rate and sample a small percentage of packets to timestamp and measure latency. Although this increases the error in measurement, it is necessary for SUME due to its hardware limitation of the number of packets to apply timestamp. The boxplot in the following graphs depicting latency measurements present details such as outlier, average, mean, median, highest/99% and lowest/1% values as mentioned in Figure 30.

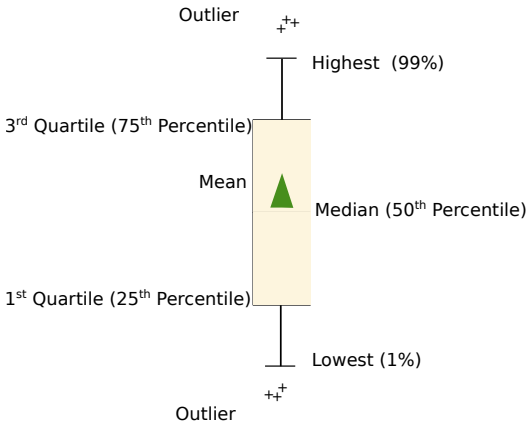


Figure 30 – Understanding Boxplot for Latency Measurements

We start this discussion with the latency results measured for the stock L2FWD example of DPDK code base for configurations with 2 CPU cores, and different burst sizes and packet sizes. The latency results are presented in microseconds (μs). The L2FWD DPDK example is carried out at 99% of the line rate and the result, in Figure 31, shows a linear increase of latency value with the increase in packet size. Similarly, we can observe that by increasing the burst size the latency of the L2FWD example also moves upward. We can observe that the latency values vary from as low as 10μs to as high as 120μs for different running configurations. Latency increases with an increase in batch size because network packets spend a longer time in the queue for packet processing in large batches. When the switch is overloaded and can not empty the queues, packets begin to be dropped, and the system ends up with a higher latency value. When we increase the packet sizes, the latency increment is nominal but linear which can be attributed to the behavior of Packet I/O that is DPDK in this scenario. DPDK might introduce latency while creating and mapping packet descriptors to represent network packets in hugepages for the packets received. Increasing the batch size boosts throughput but raises latency because the packets spend a longer time queued if processed in larger batches. This experiment is performed to understand the behavior of the testbed and how latency values change with different burst sizes and packet sizes. L2FWD is a suitable candidate for this evaluation to establish a reference because this example is the simplest example from DPDK with little to no additional delay incurred during packet processing. We will analyze and understand the results for MacS concerning this reference behavior further

in this section.

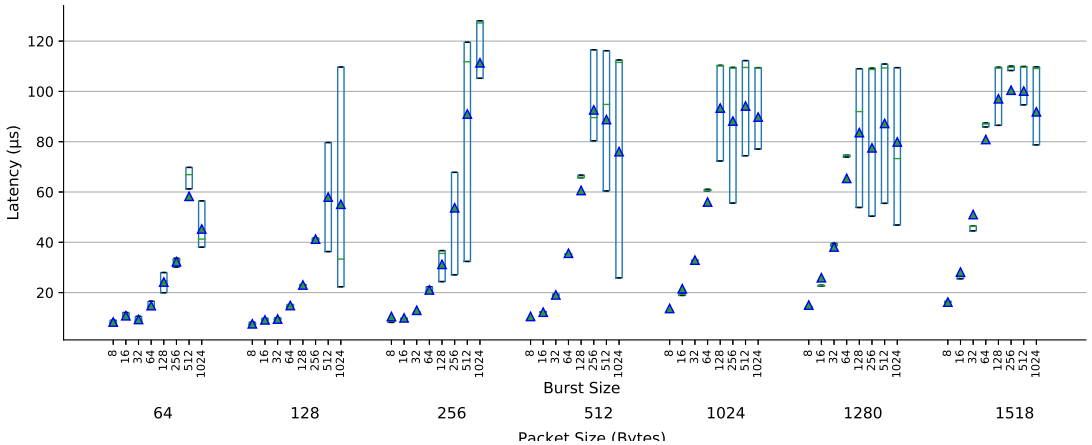


Figure 31 – Latency of L2FWD DPDK Example for different packet sizes & burst sizes. (2 CPU, 99% line rate) on Testbed E

Figure 32 presents the latency measurements for the MacS L2FWD use case over DPDK Packet I/O running with 100 FIB table size and 2 CPU cores. The different configurations consist of combination of packet sizes and burst sizes as done in Figure 31. The test traffic is maintained at 99% of the line rate for this experiment. We observe that latency increases with an increase in burst size, a behavior similar to what we observed as reference behavior in case of L2FWD DPDK example. The increase in latency value is subtle, never the less present. Further, we see that latency of 1518 Bytes packet size is smaller than 64 Bytes packet size. Also, the lowest and highest latency value are around 10μs and 24μs respectively. In fact, the highest mean latency value for MacS L2FWD is around 18μs which is way lesser than 115μs, the value of L2FWD DPDK example. After investigating the source code of both use cases thoroughly, we identified the probable cause for this difference in behavior.

L2FWD DPDK example performs the packet RX in a similar fashion to MacS, but it differs in the way it transmits out the packet after processing. L2FWD DPDK example employs two conditions when the already processed packets are transmitted out via outgoing interfaces. It forwards the packets when the TX Drain timer set to 100μs expires or if the output queue of the network interface maintained by DPDK becomes full. Hence in our scenario, as 2 CPU cores are used, two output queues are created and mapped to the CPU cores. During packet forwarding, if a queue becomes full, then DPDK will transmit out the packets from that output queue only while the other output queue will wait till it became full. Hence if the packets are distributed evenly across the cores, then we need at least two burst of packets before both the cores can transmit out packets from both the output queues resulting in additional delay. If the arrival packet rate is slower which is the case of larger packet size (1518 Bytes), then L2FWD DPDK example

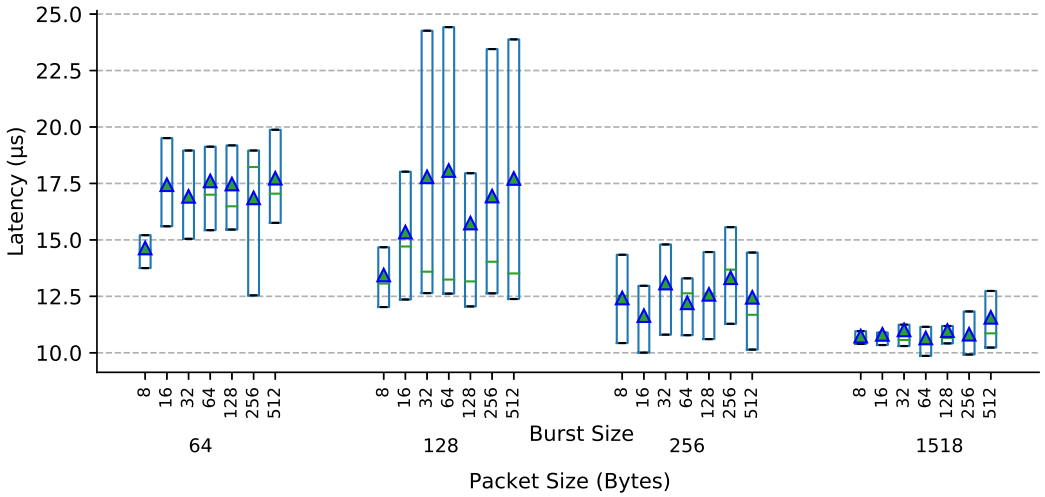


Figure 32 – Latency of L2FWD Use Case for different packet sizes & burst size. (100 Entries, DPDK, 2 CPU, 99% line rate) on Testbed E

will wait for 100μs before forwarding the packets out. Because of the delays introduced by the timer, and the flushing of the packets in 2xBatch size increases the mean latency for L2FWD DPDK example.

On the other hand, MACS employs a different approach for transmitting packets out. MACS threads mapped to a specific CPU core will receive a burst of packets each time, process the packets and then put them in the output queue buffers maintained by MACSAD. Once the processing of a batch of packets is complete, MACS will flush out all the buffers sending the whole batch of packets. In addition to that, each output queue buffer flushing also flushes all the other buffers maintained mapped to other CPU cores as well. This brings down the mean latency for MACSAD to a minimum.

To verify our understanding and validate our analysis we implemented the L2FWD MACS use case similar to L2FWD DPDK example by modifying the MACSAD source code. The results are presented in Figure 33. It can be observed that the L2FWD use case presents latency values varying between a bigger range similar to the DPDK example. Looking at this result, we can safely assume that our analysis was correct about the pattern the latency value takes for different switch configurations in DPDK L2FWD example and L2FWD MACS use case.

Another observation from Figure 33 is that the latency values are way larger than what we have seen in Figure 31. We have already seen in previous sections that L2FWD use case does not saturate the 10G NIC whereas L2FWD DPDK example is capable of functioning at line rate with minimum number of CPU core. Hence, for MACS the switch remains overloaded as the number of packets processed is less than the line rate. Due to

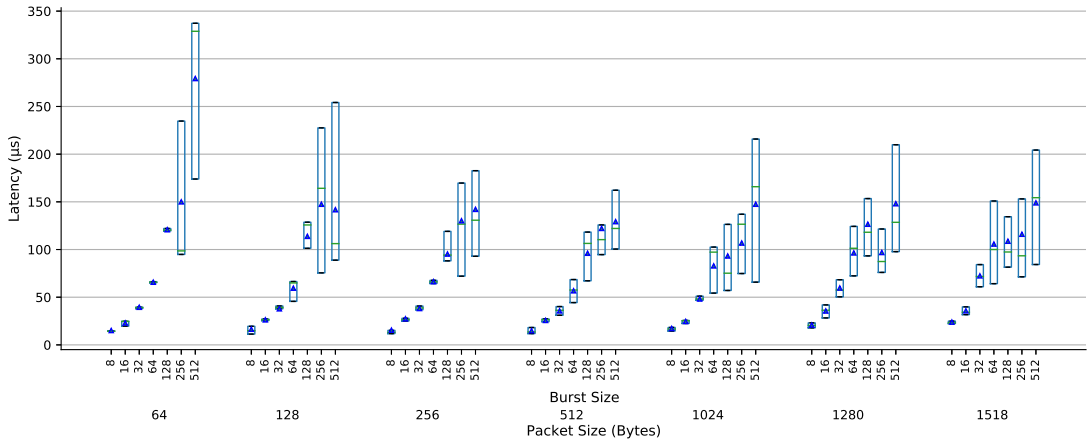


Figure 33 – Latency of L2FWD Use Case (TX part re-implemented similar to DPDK example) for different packet sizes & burst size. (100 Entries, DPDK, 2 CPU, 99% line rate) on Testbed E

the insufficient processing resources, the output queues will fill up fast, and MacS will start dropping packets due to unavailability of output queue buffers. This results in the worst case behavior for MacS and the latency value climbs up fast.

For next we moved our focus from worst case to best case scenario for latency measurements and calculated the latency values for a number of different use cases. We also bring in the Forwarding Information Base (FIB) into the mix by showcasing latency for FIB sizes ranging from 1 to 100K. The results of this activity are presented in Figure 34. The best case scenario signifies that we receive network packets at 10% of the line rate to avoid overloading the switch. Due to the lower volume of packets, packet drop due to queue buffer full scenarios is near to none. This being said, we can safely state that the results in Figure 34 represent the best results for different MACSAD use cases. The results shown in the figures are presented as multiple groups each signifying different FIB size. And each group depicts latency results in Boxplot format (Figure 30) for different use cases in decreasing order of use case complexity, i.e., BNG, NAT, L2FWDv4, L2FWD. The figure is divided into two parts where left part shows the latency for minimum packet size while the right side shows the results for maximum packet size. We identified three different expected patterns in the latency results according to the analysis presented before.

To begin with, we observe that the latency increases with an increase in packet size. This behavior is inherent to the Packet I/O and unavoidable. It also confirms the reference behavior we observed before.

Then looking into the results for different FIB sizes, it is clear that mean latency value increases in a smaller percentage with an increase in FIB size. Impact of FIB size is not as significant as the impact of packet size as seen in the figure. The low latency value is attributed to the fact that the system remains underused with traffic constituting only

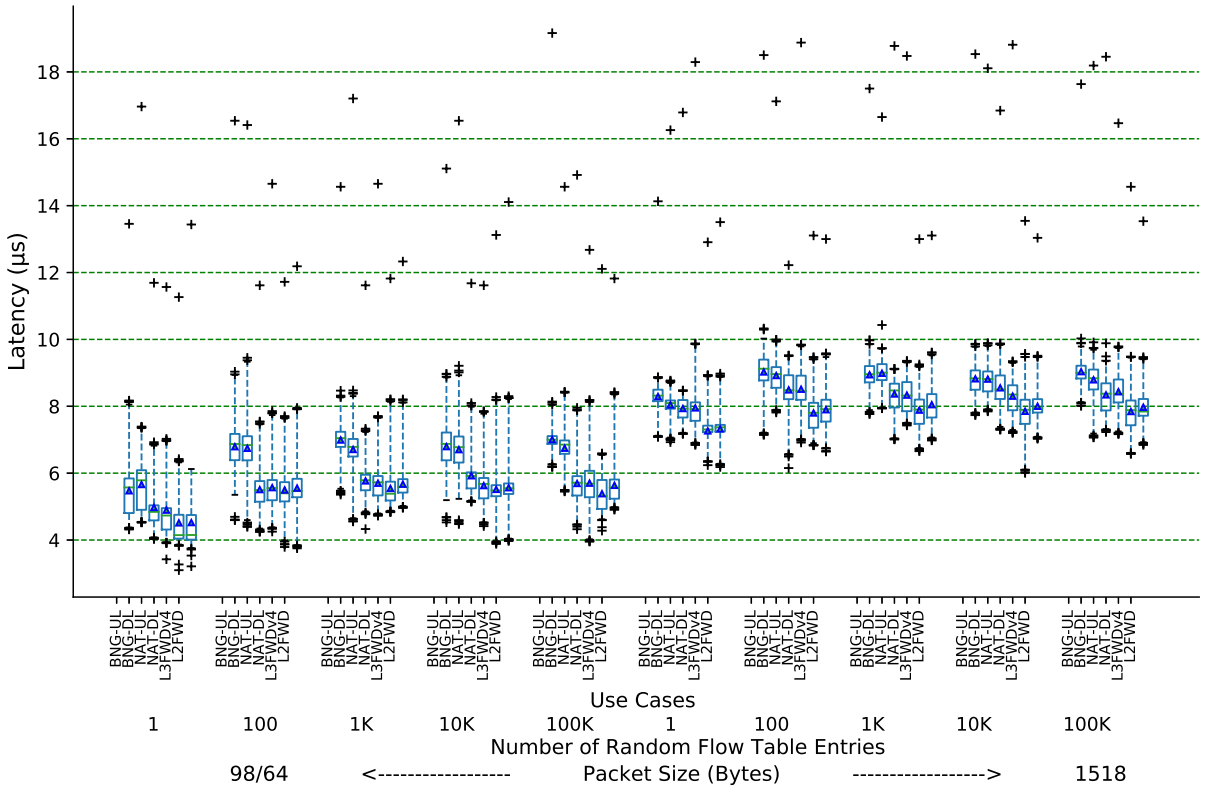


Figure 34 – Latency of different Use Cases, packet sizes, Fib Sizes. (64 Bytes and 1580 Bytes, 100 Entries, DPDK, 10% line rate) on Testbed E

10% of the line rate. Hence the increase in packet processing time taken by MacS due to the delay incurred by table lookup adds to the latency value seen in the figure.

Then we discuss the impact of use case complexity on latency. According to the Table 14 the packet processing time taken by MACSAD increases with increase in use case complexity. In the current underutilized system with 10% traffic load, the latency results will vary according to the time taken to process a packet by the use cases. Hence we see the pattern of decreasing latency value with decreasing use case complexity in the Figure 34.

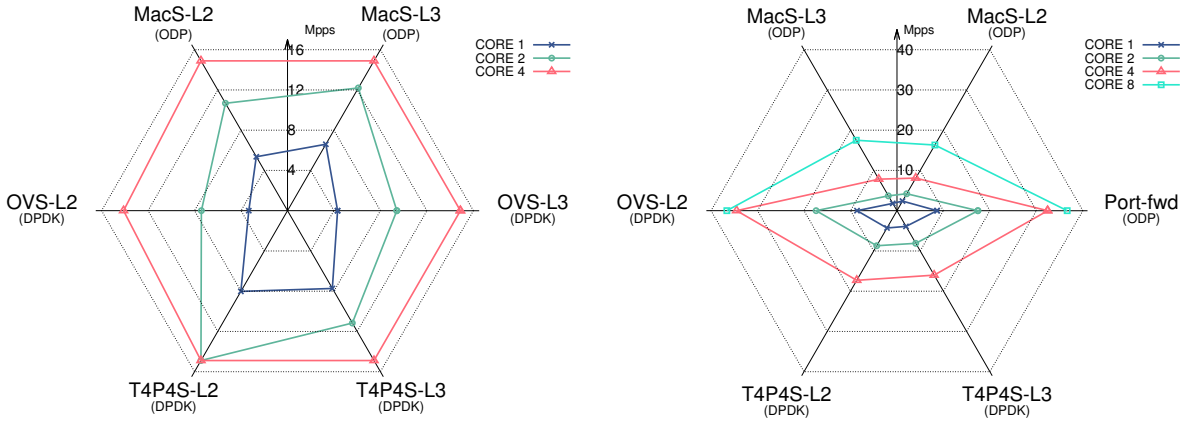
Table 15 – Latency of BNG-UL use case for different FIB sizes & packet sizes in Testbed E

Packet Sizes (Bytes)	Number of FIB Entries					% Increase (1 to 100K FIB)
	1	100	1k	10k	100k	
98	5,62	5,93	6,32	6,86	7,34	30,70%
256	5,85	6,33	6,39	7,46	7,52	28,56%
1024	6,92	7,03	6,89	8,31	8,56	23,63%
1518	8,15	7,90	7,69	8,72	9,07	11,37%
% Increase (98 to 1518)	44,96%	33,10%	21,81%	27,04%	23,53%	

To confirm our analysis and findings we looked at the percentage difference in latency values between L2FWD and BNG-DL use cases for minimum and maximum packet sizes shown in Figure 34. For minimum packet size, the difference comes to a decrease of 31% whereas for maximum packet size (1518 Bytes) this percentage decrease in latency comes down to 13%. In addition, we also explore the latency of BNG-UL use case in detail in Table 15. This table shows that the percentage increase of latency from 1 to 100K FIB entries for smallest and highest packet sizes are 30.70% and 11.37% respectively.

4.3.3 Performance Comparison Against Related Works

Achieving high performance from commodity-off-the-shelf (COTS) servers is challenging despite advances in I/O acceleration (e.g., DPDK) technologies as the presence of multiple abstraction layers (e.g., hypervisor, libraries) prevent to access all hardware capabilities (e.g., CPU, NIC). As MACSAD is developed over ODP which brings another layer of abstraction, we set out to evaluate MACSAD against other related software switches. In addition to that to assess *portability*, we expand the evaluation of the selected switches to multiple platforms, in particular to the AARCH64-based Cavium Octeon (48 cores at 2.0 GHz, shared L2, no L3 cache, and 40G interfaces). We identified T4P4S, a DPDK-enabled P4 based software switch, and OvS, a DPDK-capable open source production quality switch to carry out performance comparison with MACSAD. Both T4P4S and OvS are highly optimized to work with DPDK packet I/O. We performed this experiment on Testbed A, B, and C to bring a diverse set of target platforms and environment into the discussion. We also evaluated MacS with ODP-DPDK variant of ODP (Table 2) which is highly optimized for DPDK Packet I/O reducing the impact of ODP abstraction layer over DPDK Packet I/O observed with reference ODP implementation (Table 2).



(a) x86_64 (Testbed A) (b) AARCH64 (Testbed C)

Figure 35 – Packet Rate comparison of different platforms and switches for selected use cases (100 FIB size) and varying CPU cores.

We compare MacS with OVS and T4P4S in *Testbed A* with 100 FIB table size as shown in Figure 35a. We observe that, for 1 core, T4P4S performs better than MacS and OvS reaching around 8 Mpps for L2FWD and L3FWDv4 use cases. With 2 cores, T4P4S reaches line rate for the L2FWD whereas OVS L2FWD managed little less than 8 Mpps. MacS behaves better for L3FWDv4 use case reaching 12 Mpps as packet rate with 2 cores. All the platforms saturate the link when using 4 cores. We note that MacS lags behind T4P4S in some scenarios and this behavior can be attributed to the extra layer of abstraction brought by ODP compared to T4P4S and OvS which uses DPDK directly. We have used the ODP reference implementation applicable across different Linux kernels and not optimized for any specific target platform or Packet I/O.

Moving on to another target platform Cavium Octeon (AARCH64 architecture) in *Testbed C*, we evaluated packet rate for the three switches as presented in Figure 35b. The figure shows the measured packet rate attained on the Cavium Octeon platform for the L2FWD and the L3FWDv4 use cases implemented via MacS (top), OVS and the baseline ODP PortFWD application (middle), and T4P4S (bottom) when using 1 (blue), 2 (green), 4 (red) and 8 cores (cyan), respectively. In our results for the AARCH64 architecture, DPDK-based switches perform better than their ODP-based counterparts because the Cavium switch only supports an old and already outdated version of ODP (v1.11.0.0) instead of the current ODP version (v1.20.0.0) missing many improvements. However, the support of DPDK version for the Cavium is up to date giving an advantage to the T4P4S and OvS switches.

To assess the raw performance capabilities, we measure the baseline ODP performance with the PortFWD application, which does nothing but forward packets from one port to the other without any table lookup (right-hand side of Figure 35b). The maximum throughput with one core is about 8.6 Mpps, around 20% less than the DPDK reference throughput (11.2 Mpps, not shown in the figure). In fact, ODP is only nearly equal to OVS-L2, which does one table lookup too. Due to the raw performance differences between ODP and DPDK in Cavium, the comparison cannot be considered fair and straight forward, nevertheless serves to illustrate how the performance scales with increasing number of cores, and portability of MacS. Both T4P4S and MacS show a performance drop of about 33% against their baseline results of ODP and DPDK. We believe that with optimized new ODP support, MacS performance shall be on par with T4P4S. Current numbers show that for the L2FWD and L3FWDv4 use cases, T4P4S outperforms MACSAD in each case around 40% on average. But during the core scalability evaluation, T4P4S failed to run even with 8 cores. On the other hand, MacS easily exploited the available CPU resources, e.g., MacS L2FWD packet rate increased from 2.4 Mpps (1 core) to 16.3 Mpps (8 cores).

Finally, we will compare the three switches on our *Testbed B* running a general

purpose server (x86_64 architecture) with Mellanox 100G NIC using MLX5 drivers. We have chosen to use ODP-DPDK variant of ODP which is an optimized software implementation using DPDK which reuses a lot of DPDK packet data structures and huge page implementation for better performance and better support & transition for applications based on DPDK polling mode driver (PMD). Figure 36, 37 demonstrates MacS packet rate results conducted for L2FWD and L3FWDv4 use case with 64 Bytes packet size and 100 unique table entries for different number of CPU Cores. The switches, MacS, T4P4S and OvS are represented with blue, green and yellow color bars in the figure.

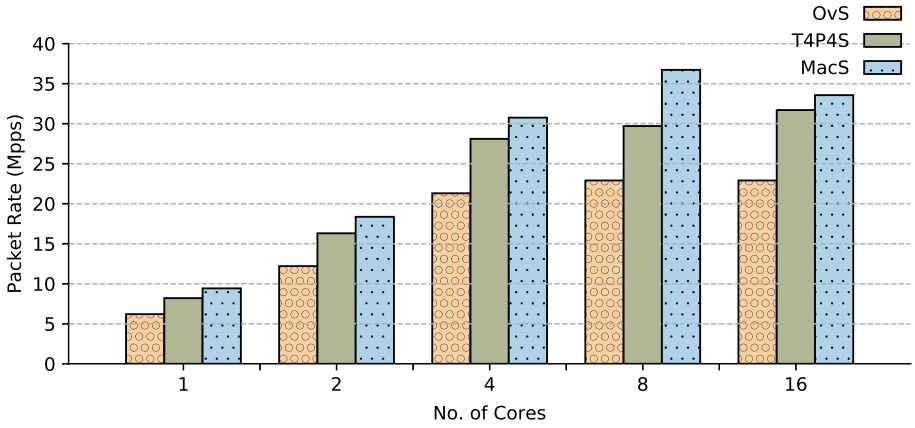


Figure 36 – Packet Rate for L2FWD (100 entries, 64 Bytes) on Testbed B.

The results for L2FWD in Figure 36 show that MacS outperforms the other two switches in case of each core configuration, and what is more, MacS scales better than T4P4S and OvS in terms of throughput while increasing the number of cores from 4 to 8. Similarly, Figure 37 shows the packet rate for L3FWDv4 use cases for 1 to 16 CPU cores. MacS achieves better packet rate compared to both T4P4S and OvS for all the core configurations as seen for the L2FWD use case.

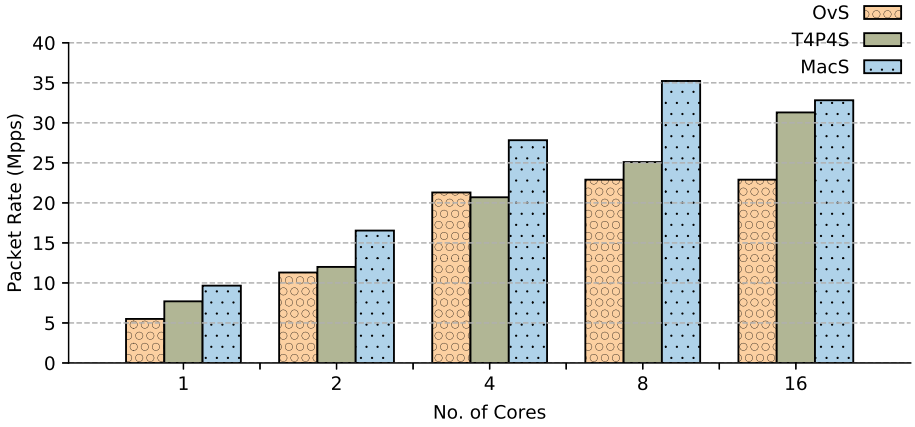


Figure 37 – Packet Rate for L3FWDv4 (100 entries, 64 Bytes) on Testbed B.

In another point, we observed that the packet rate of MacS actually decreases when moving from 8 to 16 cores. After some investigation, we attributed this loss of packet

rate to MacS mapping the packet processing threads to the new CPU cores on a different NUMA node. In the case of ODP (i.e., MacS), special attention is needed for the CPU core affinity setting when exploiting the NUMA architecture as in MACSAD we perform automatic CPU core pinning. In the Testbed B, while using more cores than 12, MacS does automatic CPU core pinning and ends up assigning cores from the other NUMA node. Memory access for a NUMA socket is always slower compared to local memory. Core allocation to remote NUMA node causes memory access delay and consequently a reduction in system performance. Despite this, the results show that even with the performance hit, MacS achieves higher packet rate compared to T4P4S and OvS while using the DPDK optimized ODP variant, i.e., odp-dpdk.

4.4 Performance Evaluation of MacS as Network Function

In our effort towards bringing MACSAD to virtualization, we tried to implement MacS with a single input and single output switch suitable as an individual network function or as part of a Service Function Chain (SFC). This is an effort to evaluate impact of packet **arbitration** on MACSAD.

The basic functionality of a packet switch is to transfer packets from input ports to output ports. Switch decides the appropriate output ports as a result of table lookup based on different packet header fields and puts the packets in the queue of the output port. When an output port queue has packets from different input ports, then switch has to decide how to schedule or prioritize the packets to send out based on the arbitration or scheduling algorithm used. Switch arbitration technique should be able to send out packets from all the input ports with minimum average latency and maximum throughput. Impact of packet arbitration can impact throughput significantly when not implemented properly. The more the number of ports in a switch, the higher the complexity of the arbitrator to maintain the switch throughput. Similarly, reducing the arbitration time can improve throughput and reduce latency across the switch. We tried to bring down the arbitration time to a minimum for our MacS as a VNF. We observed that VNF implementation follows the general practice of using one port each for both input and output. With under 50 lines of code changes to MACSAD we are able to update our MacS to operate only with one input and one output port. Due to the absence of a switch fabric, it is more relevant for a software switch similar to MACSAD to address arbitration in the software intelligently.

Figure 38 shows the observed packet rate of MacS for different number of cores with the smallest packet size, i.e., 64 Bytes. The figure presents results for L2FWD, L3FWDv4 and L3FWDv6 use cases with 1, 2, 4, and 6 cores for both MACSAD regular code MacS and the VNF optimized code MacS *(VNF)*. It is clear from the figure that

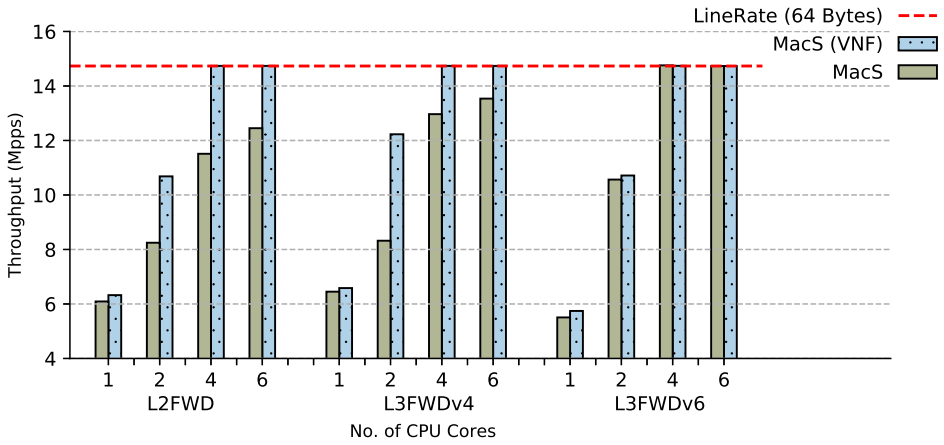


Figure 38 – Forwarding performance for different MacS (VNF) Use Cases with different CPU Cores (64B, Testbed A).

VNF optimized code performs better than the regular code base. We observed that the improvement in packet rate becomes more prominent with multiple cores in use. As per our experiment, only L3FWDv6 use case showed a smaller improvement when compared to other use cases. The red dotted line shows that MacS is able to reach line rate with 4 and 6 cores when optimized for VNF which was not the case before.

In Figure 39 we present the observed performance for different FIB sizes and packet I/O drivers. The figures show the packet rate in Mpps (left) for both L3FWD4 and L3FWDv6 VNF optimized use cases with 4 CPU cores. For L3FWDv4 use case (left), it can be observed that MacS with DPDK saturates the 10G interface even with the smallest packets (64 Bytes) irrespective of the FIB table size. Lower yield for Netmap with 64 Bytes and 128 Bytes packets confirms to previous literature [14]. Notable, the measured results for 1K FIB entries are better than for 100. This is caused by the suboptimal use of the CPU queues with small packet sizes (64 Bytes) and by the number of packets as observed in Figure 39. As expected, the Linux Socket_mmap driver stands last and never saturates the 10G interfaces.

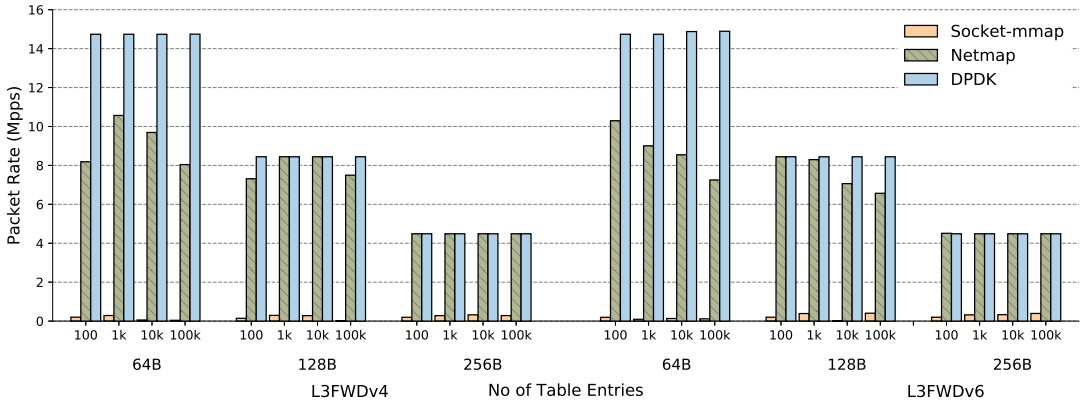


Figure 39 – IPv4 and IPv6 forwarding performance of different I/O drivers, FIB size, VNFs (4 CPU cores).

Similarly in Figure 39 (right), performance results for L3FWDv6 are on par with the findings of L3FWDv4. While DPDK reaches line rate with 64B packets for any FIB size, Netmap performance drops with increasing FIB size, and in turn, larger table size due to higher TLB misses. However, DPDK keeps TLB misses in control by using Hugepages. Also noteworthy, the anomaly for 100 and 1K FIB entries observed for L3FWDv4 does not apply for L3FWDv6.

4.5 Adaptive Scalability by Dynamic CPU Core Allocation

The importance of optimal resource utilization is more pivotal than ever due to the ubiquitous presence of virtualization in SDN environments. With MACSAD, we dig deeper into this challenge to explore CPU core allocation and utilization pattern and behavior. As seen multiple times with different use cases, MacS achieves higher performance using multiple cores where each core is responsible for packet processing of a mapped RX queue. While more cores improve performance, in case of over-dimensioning, CPU core pinning and fixed allocation to packet processing can be considered as a waste of resources. We now investigate the feasibility of a proof of concept technique to provide dynamic CPU scaling through run-time (de)allocation of CPU cores to the packet processing tasks, i.e., ODP worker threads in case of MACSAD. The decision of scaling up/down remains unexplored for now which could be adaptive based on system load, or performance measurements depending on traffic workload, or other factors (e.g., energy consumption). Such an adaptive behavior would make the system more efficient, especially in a multi-tenant environment, where de-allocated CPU cores could be used for other tasks, be them packet-processing oriented or not.

The adaptive CPU scaling technique under evaluation consists of dynamically setting the number of RX queues and accordingly scaling up or down the number of cores. To scale down, MacS removes core-queue associations, releasing the core for kernel usage and leaving the RX queue without a descriptor. A descriptor can still exist even if the corresponding queue does not. Similarly, in need of more resources for faster packet processing, our adaptive CPU technique can scale up seamlessly acquiring more cores and assigning them to the RX queues.

4.5.1 Results Analysis

We implemented and evaluated the proof of concept over two different testbeds: the First experiment runs over *Testbed A* with 4 CPU cores and INTEL x540 NIC of 10G throughput capacity, the Second experiment runs over *Testbed B* with 7 CPU cores and Mellanox MLX5 NIC of 100G throughput capacity. The two experiments explore the impact of the number of CPU cores and also the type of NICs in use. We discuss

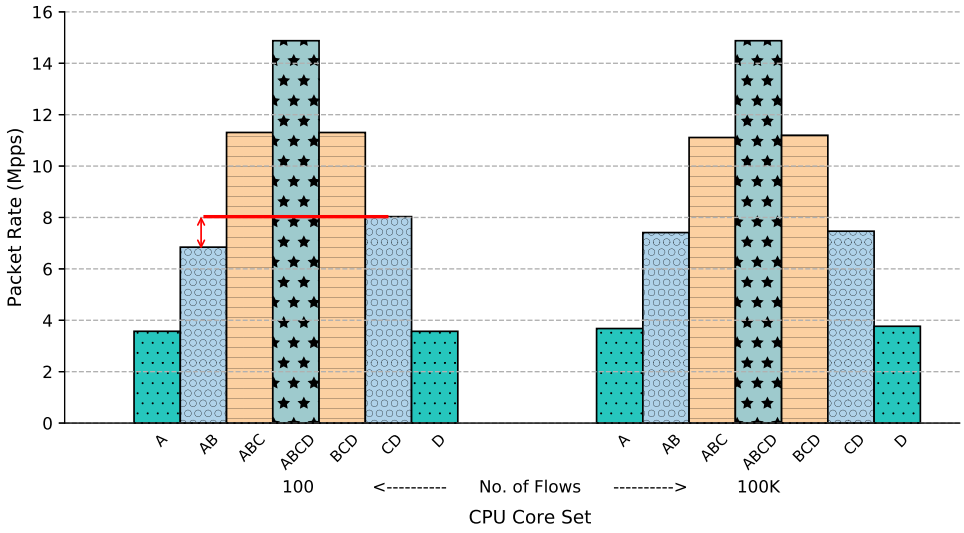


Figure 40 – Performance (Mpps) when dynamically (30s intervals) changing the sets of CPU cores allocated to packet processing for different FIB sizes on Testbed A.

the logic behind the core (de)allocation and present the packet rate details for all the configurations.

For the first experiment, we use 4 cores (`A`, `B`, `C`, `D`) and 4 RX queues, and run with L3FWDv4 use case and different FIB sizes (100, 100K) on the *Testbed A*. We set MacS to start with 1 core (`A`) and every time the 30 sec timer expires a new core is allocated (`B`, `C`, and `D`, respectively). After reaching the maximum core configuration (i.e., 4), MacS starts releasing cores, again in 30 secs interval. We use NFPA traffic generator to send test traffic (100 flows of 64 Bytes packet size with unique 5-tuple headers) at line rate (10G) to overload the DUT.

Figure 40 shows how the obtained throughput increases and decreases in line with the number of active cores. Although the results presented are an average of 10 different runs for traffic with 100 and 100K unique number of flows, the observations were consistent over different run. Figure 40 shows that the CPU core set (`AB`) achieves lower throughput compared to (`CD`) when the number of flows is 100. Since the sending rate was fixed throughout the experiment, the only explanation is the RX queue receiving less traffic over (`AB`) compared to (`CD`) core set and not a limitation of MacS TX queues. Under an ideal traffic distribution with both of the two-queue/two-core sets, we should observe the same throughput as in case of the (`ABC`) compared to (`BCD`) allocation, or when only cores `A` and `D` are used. The unequal flow distribution observed for the traffic trace with less number of unique flows could be explained by specificities of the Receive Side Scaling (RSS) hashing function implementation and the statistical nature of such load-balancing mechanism and hence the challenge of always deciding on the optimal number of CPU cores for a certain throughput requirement for the target platform. Hence, the obtained

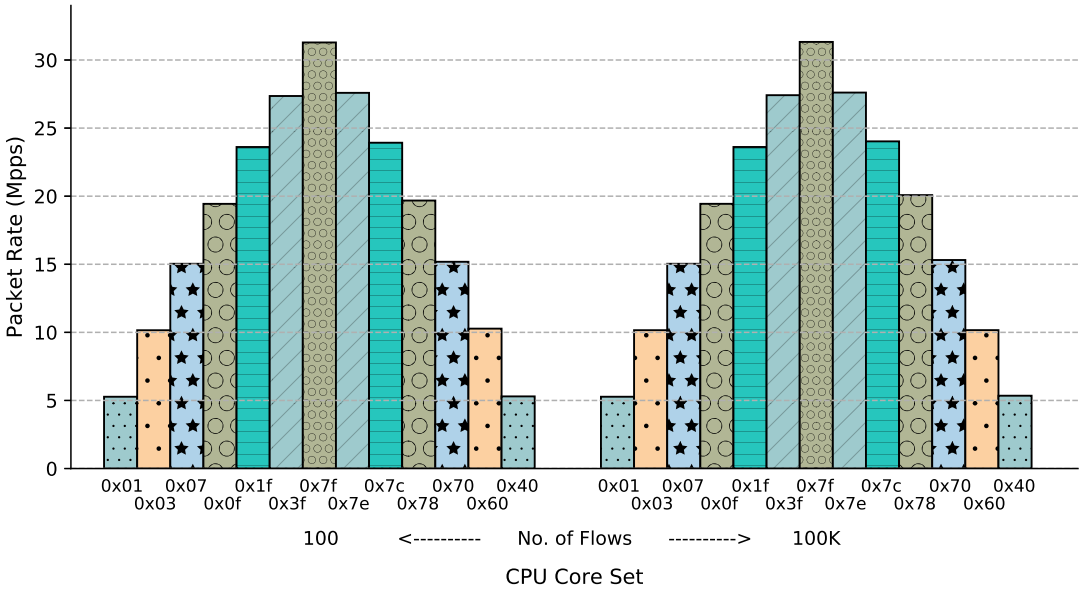


Figure 41 – Performance (Mpps) when dynamically (30s intervals) changing the sets of CPU cores allocated to packet processing for different FIB sizes on Testbed B.

results found evidence regarding unequal traffic distribution behavior of RSS when only 100 flows are balanced through the cores. By increasing the FIB size to core ratio, with 100K FIB size, we can avoid this behavior on our Testbed with INTEL x540 NIC.

As part of the Second Experiment, we verify the RSS behavior on a production-grade NIC and a higher number of CPU cores for this use case over the *Testbed B* with 100G Mellanox smart NIC and 7 CPU cores respectively. This provides us with a base to see if our solution is able to scale or not in terms of raw throughput and also with more CPU cores. We can use more CPU cores, 7 CPU cores to be specific, instead of only 4 CPU cores as the available NIC with 100G line rate does not get saturated as would have been with a NIC of 10G line rate as in the case of the previous experiment of *Testbed A*. Figure 41 shows how the obtained packet rate increases and decreases in line with the number of active cores available. We also took an average of 10 different runs for each FIB size for this experiment. The CPU core (de)allocation is done similar to Figure 40 with every 30sec. The active CPU cores are presented in CPU Mask format on the x-axis. We found that the RSS performs in a near ideal manner and able to distribute the traffic across RX queues evenly giving a symmetrical convex shape to the graph in Figure 41. As a result, we are able to show how MACS throughput/packet rate increases and decreases in a deterministic way with allocation and de-allocation of CPU cores.

4.5.2 Discussion

Dynamic CPU Core Allocation activity focuses on the idea of Resource Scaling introduced in subsection 1.3.2. Exploring the outcome of two experiments running on dif-

ferent testbeds with a different number of CPU cores, and different type of NICs broadening the evaluation breadth, we show how MACSAD behaves robustly in each case. As an unexpected outcome to this activity, we detected an issue with some NICs (e.g., Intel 82599, X540) which stopped receiving packets during the experiment. In particular, when an RX queue is not fully flushed before removing its RX descriptors by de-allocating the associated CPU core, the NIC stops processing packets altogether from all RX queues whereas other NICs (e.g., Intel XL710 Fortville) did not have this limitation allowing us to execute this experiment successfully. We have reported this issue to the ODP community[3] and helped to resolve it towards our effort to contribute to the open source community.

4.6 Concluding Remarks

Following the discussion around the design and implementation of MACSAD, this chapter took a direction towards the evaluation of flexibility in MACSAD. We extend the discussion about evaluation of MACSAD into the different aspects of flexibility across categories introduced in subsection 1.3.2, i.e., *Programmability, Portability, Performance & Scalability (3PS).*

We started this chapter by explaining the testbeds used in section 4.1 and the different use cases supported in MACSAD in section 4.2 exploring the *Portability* and *Programmability* aspects respectively. We then evolved the discussion addressing *Performance* in terms of *Packet Rate* and analyzed the impact of FIB sizes, Burst sizes and also different Traffic Generators in section 4.3. We bring latency into the discussion as another aspect of *Performance* and explored in subsection 4.3.2. Discussion around related works in subsection 4.3.3 compared MACSAD against other related projects like OvS and T4P4S. In section 4.4, as a step towards virtualization support we showed how MACSAD could act as a VNF with preallocated resources leveraging *Programmability* and *Scalability* aspects. We identified resource scalability as another aspects of flexibility beyond 3PS, and explored the same in section 4.5 by evaluating on-demand resource scaling in terms of CPU cores in MACSAD. Meanwhile, flexibility in terms of design introducing compiler optimization with minimal effort (chapter 6) is discussed in a later chapter.

With the knowledge about the use cases and their performance details, we proceed to the next chapter 5 which explains a methodology to represent the complexity of different use cases and also demonstrates the application of machine learning to produce a complexity model of MACSAD predicting the performance of the use cases.

[3] <https://bugs.linaro.org/show_bug.cgi?id=3618>

5 Complexity Analysis

We dissertate the complexity of all the use cases in this chapter and present a methodology to represent use case complexity in a more quantifiable manner. We follow up the discussion by bringing machine learning algorithms to create a performance prediction model using complexity features.

5.1 Use Case Complexity

We have explored different use cases in detail in chapter 4. We mentioned that the use cases vary in terms of complexities of their pipelines. The traditional view of switch pipeline complexity is not sufficient enough to describe a P4 defined pipeline due to its difference in abstraction level. Hence a new methodology is necessary which can explore different P4 constructs identifying the abstractions to calculate the complexity. We took inspiration from (DANG *et al.*, 2017) while identifying the constructs from the P4 program to define the complexity of the P4 based pipeline. Table 16 presents the constructs labeled as *Complexity Factors* from P4 programs classified into different categories. We have identified seven categories of *Complexity Factors* and explored six of them in our use cases. According to the P4 abstraction model shown in Figure 4, the P4 based switch consists of 3 stages, i.e., Parser, Table & Lookup, and Actions. Our categories of *Complexity Factors* broadly fall into those stages with an additional category for Stateful parameters. We have excluded the Stateful P4 constructs due to their lack of support in MACSAD and absence in our use cases in current shape. The different categories of *Complexity Factors* are as follows:

Parsing.

Upon arrival of a packet in a switch, the parser extracts the headers and header fields, and update the relevant metadata. Parser stage is expressed by the P4 abstraction named *Parser* and represented as a finite state machine in MACSAD. Network packet parsing overhead increases with increasing number of headers and/or header fields to be parsed, and presence of branches in a parse graph. Branches appear in a parse graph when the parser needs to check one or more header field values to transition into another parse state to initiate parsing the next header specified in the P4 program. Hence *Parsing* category has three complexity factors where Packet headers and header fields express the total number of headers and header fields respectively, while the *branches* emphasizes the conditional parsing of up next header based on the current header field values. The parser branch details can be easily understood by the parser graph shown in the Annexes (e.g., L2FWD [Appendix A], L3FWD [Appendix B], etc.). For L2FWD use case, only the

Ethernet header is parsed and hence the *Packet Headers* has value as one in the table. Similarly, L3fwd has a value of 2 for Ethernet and IP headers involved. To continue, DCG-UL and DCG-DL use cases have values 6 and 3 respectively. For the DCG-UL use case, MacS receives network traffic with VXLAN encapsulation resulting in higher number of headers to be parsed compared to the DCG-DL use case.

Processing.

MacS uses tables to process the network packets on the completion of the parsing stage. P4 abstractions *Table* from the P4 control flow is responsible for this category. We identify the total number of tables and the pipeline depth (i.e., the maximum number of tables with dependencies) as *Complexity Factors*. MACSAD pipeline processing dictates how the network packets interact with the tables. Although P4 does not mandate to have tables, it is necessary to have at least one table to do any kind of packet processing in a P4 program. Here the dependencies among tables can be explained as the scenario when processing of a network packet is transferred to a new table depending on the outcome of *match+action* at the current table. This allows a packet to skip one or more tables in the P4 program pipeline when necessary. Hence the depth of a pipeline can be calculated as the number of tables a network packet is handled by for a specific protocol or use case pipeline. The depth of a pipeline is equal or less than the total number of tables in a P4 program. Looking at other complexity factors, size of tables and if a checksum is necessary in the tables are also identified as important *Complexity Factors* under this category. DCG and BNG use cases have a higher depth of pipeline being more complex use cases. On the contrary, PORTFWD in subsection 4.2.1 is a use case designed to have zero complexity and hence have no tables defined in the pipeline.

Lookup.

This *Complexity Factors* category extracts the lookup details from *Table* P4 abstraction. We only show the Hash and the LPM based lookup in the Complexity Table 16 as currently MACSAD only supports EXACT (hash-based) and LPM lookup types from the P4 specification. Lookup event being one of the costliest events in the switch pipeline, we chose to assign a separate category for it. The Complexity Table is populated with the number of lookup operations, and length of the lookup keys for each use case. For example, 2[48] value as the entry for L2FWD use case signifies the presence of two hash-based lookup operations with lookup keys of size 48 bits each.

Header Update, Field Update, Metadata Update.

MACSAD acts on the packet headers and metadatas according to the Actions defined in the Tables while processing network packets. These Actions are also identified as P4 constructs or abstractions present in the P4 program. We divide this P4 abstraction into three separate categories: One showing all the add/remove/copy header operations;

Another highlighting the metadata update operations; Finally the last one focusing on all the header field updates. Separating the P4 Action construct into three separate categories is essential due to the different nature and impact on the performance of these three operations. Metadata is unique as it is stored separately from packet structure while has a life span equal to the packet. MacS implements P4 defined standard metadata and also supports user-defined metadata defined in the P4 program. INGRESS_PORT and EGRESS_PORT metadata are the standard metadata common for all the use cases in MACSAD and use extensively in the MACSAD source code. Going forth, header add/remove requires multiple memory accesses and multiple header field updates pushing the use case pipeline towards memory bound. Hence *Header Update* deserves a separate category. Similarly by creating a separate *Field Update* category, we can enforce a clean separation between Header related updates and Header Field modification operations, and also acknowledge the impact of field modification operations over MacS performance.

State Accesses.

State Accesses is the final category shown in Complexity Table 16 with 4 different complexity factors. These complexity factors are based on the stateful operations supported by P4 specifications. Most of the P4 constructs we have explored under MACSAD are stateles as these parameters produce results solely based on inputs given to it. Similarly, $P4_{16}$ defines two types of stateful constructs which are capable of retaining values across packets. *Table* construct is the first stateful P4 construct which is read-only and can only be modified by the control plane. The other stateful construct *EXTERN* Object is the construct we focus on in our discussion. This *EXTERN* Object can be read and modified by data plane. In $P4_{14}$, this construct is represented as three different construct types such as counters, meters, and registers. P4 allows read and write to registers, which is the most commonly used stateful construct, to perform stateful operations. Hence we have identified four different types of complexity factors under *State Accesses* depending on the differences in underlying operations: Write to Different Registers, Write to Same Register, Read from different Registers, & Read from Same Register. As use cases under the umbrella of MACSAD do not implement any stateful operations, we have ignored more detailed exploration of this category in our analysis of use case complexity, and the Complexity Table 16 has values 'zero' for all the Stateful Complexity Factors.

Complexity Table 16 shows details of all the complexity factors for every use cases explored in section 4.2. From the table, it is possible to grasp the difference in pipeline complexity between different use cases looking at different complexity factors. We want to use this data to further understand the behavior of MacS in-depth under different configurations of MacS. Our motivations are to develop a mathematical model using the complexity factors which can predict the performance of MacS on demand. These ideas are explored in the following section of this chapter.

Table 16 – P4 Use Case Complexity Details

		L2FWD	L3FWDv4	L3FWDv6	NAT-UL	NAT-DL	DCG-UL	DCG-DL	BNG-UL	BNG-DL
		$P4_{14}$	$P4_{14}$	$P4_{14}$	$P4_{14}$	$P4_{14}$	$P4_{14}$	$P4_{14}$	$P4_{16}$	$P4_{16}$
Parsing	Packet headers	1	2	2	3	3	6	3	5	3
	Packet fields	3	13	19	16	16	36	13	46	25
	Branches	1	2	2	3	3	6	2	4	2
Processing	Total no of tables	2	2	2	6	6	10	10	9	9
	Depth of pipeline	2	2	2	5	5	7	8	6	6
	Checksum on/off	off	off	off	off	off	off	on	off	off
Lookup	[Hash Based] key width (in bits)	2 [48]	1 [9]	1 [9]	1 [9] 1[48], 1[16]	2 [9] 1[48], 1[16]	2[48], 3[32], 1[9]	3[48], 2[32], 1[2], 1[9]	2 [9] 1[32], 2[48]	1[1], 2 [9] 1[32], 1[48]
	[LPM Based] key Width (in bits)	0	1 [32]	1 [128]	2 [32]	1 [32]	1 [32]	1 [32]	1 [32]	1 [32]
Header Update	Header adds	0	0	0	0	0	0	4	0	3
	Header removes	0	0	0	0	0	4	0	4	1
Metadata Update	Metadatas	1	1	1	1	1	2	3	12	24
Field Update	Field writes	2	4	4	5	5	11	24	14	26
	Arithmetic expressions	0	1	1	0	0	0	0	0	0
	Boolean expressions	0	0	0	0	0	0	0	0	0
State Accesses	Write to different register	0	0	0	0	0	0	0	0	0
	Write to same register	0	0	0	0	0	0	0	0	0
	Read to different register	0	0	0	0	0	0	0	0	0
	Read to same register	0	0	0	0	0	0	0	0	0

5.2 Machine Learning (Regression) Analysis

Machine Learning (ML) with numerous algorithms under its umbrella is useful to solve problems like classification, prediction, etc., where it takes a dataset as input and learns from the data. As a standard practice, the dataset is divided into two part: Training & Test Data. Training Data is used to train the ML system; then the Test Data is used to verify and validate the trained ML system. When the dataset fed to the ML algorithm contain the desired solutions i.e., *labels/measured values*, then the ML system is known as *Supervised Learning*. Similarly, in *Unsupervised Learning*, the data set is unlabeled, and the ML system tries to learn automatically during the training of the system. As part of the analysis of the performance results of the MacS, we already have the *labels* in our data set dubbed as *Packet Rate* expressed in Millions Packet Per Second (Mpps). With labeled dataset, we use a model-based supervised learning ML system as our choice of ML system. We expect to learn from the dataset by generalizing the input to build and train a model also know as to fit a model. Then, use the trained model to predict the performance in terms of packet rate for test data and also for new inputs.

The two integral part of any ML system are *Dataset* & *ML Model*. For this activity, the dataset is collected by running MacS on our **Testbed D**. We gathered the packet rate of MacS for different use cases with different configurations. We selected packet rate of L2FWD, L3FWD and NAT use cases as the base pipeline, and then we modified these pipeline by changing the number of parsers or parser branches, changing the number of tables, modifying lookup types and varying the number of header fields and metadata update operations in the respective P4 programs. Only these complexity factors from the Complexity Table (Table 16) are considered for this activity. The different complexity factors in the dataset used for learning by the ML system are known as *features* or *predictors* as they are used to build, train and fit the prediction model. The packet rate calculated for each run of MacS is the *label* or *predicted value* for the dataset. The values of features are extracted from the P4 program as specified in the Complexity Table (Table 16). Apart from these, there are other target specific factors affecting performance which are also considered. However, for brevity, we have considered only one target specific factor, i.e., Number of CPUs, while ignoring others (e.g., CPU frequency, RAM available, Hardware Acceleration available, etc.). By restricting our experiment to a single Cavium target we can confirm that the missing target specific factors do not bring any bias to the dataset. We used our Cavium bare-metal switch with ThunderX SoC for data gathering due to the consistent performance of MacS over this target across runs and use cases. We mean our ML model to fit with features identified by the complexity factors extracted directly from the P4 program which out any compilation needed. This provides us with an opportunity to predict the performance rate for a new MACSAD use case represented by a new P4 program without even compiling and running MacS on the target device. It will undoubtedly help to automate or at least facilitate in decision-making whether to deploy a specific use case over a specific target device when the scenario is sensitive to packet rate.

A sample of our dataset collected from MacS executions with different pipelines is shown in Table 17. There are six features shown as the first six columns in the table. The last two column shows the Packet Rate in million packets per second (Mpps) and mega bit per second (Mbps) respectively. Features are the independent variable and packet rate is the dependent variable as packet rate depends on the input features. We choose to work with *Regression* based ML models which are suitable for prediction problems and small dataset with a fewer number of features. We analyze MACSAD dataset by applying a simpler multivariate Linear Regression model and also different complex regularized linear models like Ridge and Lasso Regression models. Different variants of Regression models are capable of handling different characteristics of the input data allowing us to decide upon the algorithm which suits our experiment and data well. For every ML model, we will have our dataset divided into Training Data and Test Data at 3:1 ratio with test data having 25% of the total data entries. We have divided our discussion into *Data Processing*

and *Regression Models* for better clarity and understanding. We bring machine learning concepts into MacS evaluation, and its use case complexity analysis to fulfill the following goals:

- To represent MacS performance in terms of a mathematical model using complexity factors.
- Identification of the degree of influence on performance by different complexity factors.
- Ability to predict MacS performance on demand by extracting features from the P4 program itself.

Table 17 – MacS Dataset Sample

CPU Count	Parser		No. of Field Updates	EXACT Count	LPM Count	Packet Rate	
	Header Count	Branch Count				MPPS	MBPS
1	0	0	0	0	0	4.50	3027
2	0	0	0	0	0	9.04	6072
3	0	0	0	0	0	13.40	9006
4	0	0	0	0	0	14.88	9999
2	2	1	4	1	1	2.45	1648
4	1	0	1	2	0	4.03	2706
1	1	0	0	0	0	4.29	2880
1	4	0	0	0	0	3.99	2680
1	6	0	0	0	0	3.82	2568
1	1	0	0	1	0	1.38	930

5.2.1 Data Processing

Collecting and processing the input dataset is an important part of every ML system. Once the dataset is available, we need to understand and curate it to be useful to our chosen ML models. There are different challenges when handling a dataset concerning quality, quantity or effectiveness of the data towards the ML models. We present some of the common challenges encountered while preprocessing MACSAD dataset; (1)Dataset Size, (2) Nonrepresentative & Poor Quality Data, (3) Feature Selection.

Dataset Size

In the world of ML, it is hard to make rigid rules or define fixed practices as every ML system is unique and different in a way to suit the problems in hand. This stays

valid while estimating the optimum size of a dataset too. However, there are some thumb rules to start with to categorize and explore different kinds of ML problems. We explored different thumb rules (VANVOORHIS; MORGAN, 2007) to understand the process of finding the perfect size of a dataset. One of the thumb rule mentioned in (GREEN, 1991) says that the sample size (N) should be:

$$\begin{aligned} N &> 50 + 8T \qquad (to\ test\ multiple\ corelation) \\ N &> 104 + T \qquad (to\ test\ individual\ Feature) \\ &(where\ T\ is\ the\ number\ of\ Features) \end{aligned} \tag{5.1}$$

Similarly, another school of thought exploring specifically multivariate scenarios (HARRIS, 2001) advises the calculation of sample size (N) as shown in Equation 5.2. According to Harris (HARRIS, 2001), for regression analysis, the dataset size should be at least 50 more than the number of features, or it should be at least ten times the number of features for the scenarios where the number of features is less than six or greater than equal to six respectively.

$$\begin{aligned} N &\geqslant 50 + T \qquad (For\ T < 6) \\ N &\geqslant 10T \qquad (For\ T \geqslant 6) \\ &(where\ T\ is\ the\ number\ of\ Features) \end{aligned} \tag{5.2}$$

The last two columns of Dataset sample at Table 17 are the dependent variables depicting the *Packet Rate*. The first six columns are the six features or also known as dependent variables available in our dataset. With six predictors available, the minimum sample size according to Equation 5.2 should be 60. Our dataset of 70 entries adheres to the thumb rule to go ahead with the regression analysis.

Nonrepresentative & Poor Quality Data

Moving onward, we will have qualitative analysis rather than quantitative analysis of the dataset. ML models generalize on the basis of the training data which is the representative of the relevant use case involved. This helps the generalized ML model to predict the packet rate values for new use cases in turn. Hence we looked into our dataset to access and identify the data entries which do not represent the use cases we want to generalize the model to. The initial 4 entries in the Table 17 are based on the PortFWD use case defined at section 4.2. We observe that apart from the *CPU Count* predictor, every other predictor have values as zero. This PortFWD use case is designed to evaluate the raw performance of the testbed and not an effective real-world use case similar to L2FWD or NAT. Hence the data entries for PortFWD are considered as nonrepresentative in our case and are removed from the dataset before training the ML models. Similarly, we also avoided the data entries to be part of our dataset where the *Packet Rate*, the dependent variable, has a value of 14.88 Mpps, i.e., the line rate of the 10 NIC of in testbed. As the dependent variable is capped at 14.88 Mpps, we can not obtain the theoretical packet

rate possible for the corresponding set of features as the hardware limits the measured value. This is considered as a type of Poor Quality data, and are not considered to be part of our dataset. Another important example of poor quality data is outliers. However, Cavium servers being consistent with performance, our dataset is relatively immune to outliers. With the removal of the nonrepresentative data, our dataset size is reduced from 70 to 66 while still an acceptable size according to Equation 5.2.

Feature Selection

Another dimension to the data quality of the dataset lies in the features selected for the ML models. It is possible to have a number of features available in the dataset, to begin with, but it is essential to identify the relevant features which have sufficient impact on the dependent variable and represent the problem statement in hand. This process of screening of features is known as *Feature Selection* or *Feature Engineering*. Feature Selection can be performed by applying different algorithm under Machine Learning or by an expert on the dataset being used. Complexity Table (Table 16) shows all the features relevant for our experiment, whereas all the features may not have the same level of relevance to our models. With our domain expertise and knowledge of MACSAD internals, we have identified the most relevant features as shown in the Table 17. We have also selected *CPU count* as an additional feature which was possible due to our domain knowledge. Similarly, irrelevant features should be discarded to avoid bias during the training of the ML model. The *Mbps* feature shown in the last column of Table 17 is one of the features we have avoided using in the ML model as it is a derivable feature which can be calculated from the *Mpps* column and does not bring any additional value to the ML model.

5.2.2 Regression Models

Regression methods estimate the relationship among features or predictors and the dependent variable. With a generalized regression model, it is possible to predict the value of the dependent variable (Y) with a change in the corresponding independent variables (X). Due to this nature, we have chosen to use regression models with the MACSAD packet rate dataset. We can represent every regression model as:

$$Y \approx f(X, \beta).$$

There are three important parameters involved here: Y, the dependent variable; X, the independent variable; β, coefficient. Although every regression model represents the dependent variable as a function of independent variables X and β, with different regression algorithms, this function changes accordingly.

We will begin with the Multivariate Linear Regression as the simplest model and its support for multiple features to generalize the model. Collinearity is a common issue with dataset when we have multiple features. Collinearity appears when a feature is correlated

with one or more other features. Collinearity may lead to an increase in the variance of the regression coefficients making them unstable. Collinearity can be a problem when the features are measured with error. But with our Cavium Testbed, we are confident about the quality of the measured values of the features, and hence collinearity should be less of a concern in our case. Nevertheless, in anticipation of collinearity, we have chosen two regularized regression models too (i.e., Ridge Regression, and Lasso Regression) which are immune to collinearity to some extent by implementing regularization to the dataset. Like collinearity, scale imbalance among the features is also another dataset characteristic we need to evaluate while applying regression models. In the presence of multiple features, the feature with a smaller scale may result in a lower impact towards the final cost function of prediction. Although feature scaling does not affect the linear regression model and only affect Ridge and Lasso regression, we apply feature scaling to the dataset in all our regression models. By doing this, we can directly compare the results of the three regression models. Once a regression model is chosen, we need to fit the model using training data before using it to perform predictions. We have identified Mean Square Error (MSE), Root Mean Square Error (RMSE), Mean Absolute Error (MAE), and Coefficient of Determination (R^2) as our preferred performance measures to evaluate and compare the different regression models. The performance measures can be understood as values calculated over the distance between vectors: prediction vector ($\widehat{y}_i$), and measured vector (y_i).

To begin with, MAE depicts the value of the absolute error loss or l_1-norm loss. It calculates the average of the absolute errors in prediction where the error is the difference between predicted ($\widehat{y}$), and measured (y) values, and hence every error contributes to MAE in proportion to the absolute value of the error. Equation 5.3 demonstrate the equation to calculate the MAE value for a model. Next, MSE and RMSE both corresponds to l_2, and are represented in Equation 5.4,5.5. MSE metric represents the average of the squared error or loss where the error is the difference between predicted ($\widehat{y}$) and measured (y) values. MSE is a non-negative quality estimate of a model, and a near-zero value is considered to be better. RMSE is calculated as the square root of MSE and corresponds to l_2 norm similar to MSE. RMSE is easily interpretable than MSE as it has the same measurement units used to fit the model. Inherently RMSE is susceptible to scale of the features used. Both MAE and RMSE are used widely for regression tasks. These metrics are negatively-oriented scores as approaching to zero means better fit model. RMSE put more weight to larger errors as it squares error before calculating the average. As a result, RMSE is more sensitive to outliers than MAE, but it performs well when outliers are rare. Another difference between them is that RMSE result will always be larger or equal to MAE depending on the magnitude of errors. Interesting to consider that, RMSE is likely to have a higher value than MAE when the size of test data increase. Hence while comparing RMSE values, it is important to maintain the test data size equal. We will use MAE and

RMSE to compare different regression models. To conduct the comparison efficiently, we have chosen to apply feature scaling and keep data split ratio between the train and test data across regression models consistent considering the nuances of MAE and RMSE. Finally, the Coefficient of Determination (R^2), the most important performance measure of regression analysis, is shown in the Equation 5.6 and is considered to be the regression score function. This represents the proportion of the variance in the measured value that can be predicted by the fit model. Otherwise, we can say that R^2 value tells us how well our model can predict for future samples. R^2 ranges from 0 to 1 where 1 means the model can predict without any error. For example, a R^2 of 0.50 means that our model can predict 50 percent of times with certainty. R^2 is mostly used to evaluate how a specific model is measured for its efficiency and accuracy.

$$MAE = \frac{1}{n}\Sigma_{i=1}^{n} \mid \widehat{y}_i - y_i \mid \tag{5.3}$$

$$MSE = \frac{1}{n}\Sigma_{i=1}^{n}\left(\widehat{y}_i - y_i\right)^2 \tag{5.4}$$

$$RMSE = \sqrt{\frac{1}{n}\Sigma_{i=1}^{n}\left(\widehat{y}_i - y_i\right)^2} \tag{5.5}$$

$$R^2(y, \widehat{y}) = 1 - \frac{\Sigma_{i=1}^{n}(y_i - \widehat{y}_i)^2}{\Sigma_{i=1}^{n}(y_i - \overline{y}_i)^2} \tag{5.6}$$

where:

n = number of samples
y_i = measured value of i^{th} sample
$\widehat{y}_i$ = predicted value of i^{th} sample
$\overline{y}_i$ = mean of the measured values
$= \frac{1}{n}\Sigma_{i=1}^{n} y_i$

Multivariate Linear Regression

Linear Regression is the simplest model from the three models we have selected for this task. Linear Regression performs prediction by computing a weighted sum of the features plus a constant *intercept* term as shown in Equation 5.7. Due to the presence of multiple features, it is also known as multivariate linear regression.

$$\widehat{y} = \Theta_0 + \Theta_1 X_1 + \Theta_2 X_2 + ... + \Theta_n X_n \tag{5.7}$$

where:

$\widehat{y}$ = predicted value
n = number of features

X_i $= i^{th}$ feature value
Θ_j $= j^{th}$ model coefficient (including the bias term Θ_0 and the feature weights Θ_1, Θ_2, ..., Θ_n)

To train the linear model we have to find the coefficients in Equation 5.7 so that the model fits the training data. A model properly fits the dataset when the RMSE value tends to minimum. Hence we need to identify θ vector which minimizes the RMSE. Then, we calculate the predictions over test data and the R^2 score of the model.

Regularized Linear Regression

Regularization of the linear model is necessary to averse overfitting of data. Overfitting means the model performs better on the training data but does not generalize well. As a result, it may predict erratically against test data or future data. Overfitting can be a result of a complex model with a large number of features compared to the dataset size. While reducing the number of features or gathering more data are a couple of solutions for overfitting, it is not practical in our case. We have carefully selected all the crucial features, and we want to build a model using all the selected features. We opt to go with constraining the model by regularization controlled by a hyperparameter to overcome overfitting. In regularization, we reduce the coefficients while keeping the number of features constant. With regularization, our model stays robust against multicollinearity among features too. We continue to explore Ridge and Lasso regression methods and their ways to constrain the coefficients as part of Regularized Linear Regression.

Ridge Regression

Ridge Regression is a regularized linear regression where the cost function is modified by adding a regularization term equal to the square of the magnitude of the coefficients ($\alpha\Sigma_{i=1}^{n}\Theta_i^2$). This regularization term is in l_2 norm. Ridge regression puts constraints on coefficients with the help of a hyperparameter (α) which controls how much we can regularize the model. The cost function of Ridge regression is shown in Equation 5.8. From the cost function, we can observe that at α equals to zero Ridge Regression becomes similar to Linear Regression. Hence we can say that higher the alpha value, more restriction on the coefficients; lower the alpha value, more generalization and the coefficients are barely restricted. When α is large, then all the coefficients approach to zero but never equals to zero, and the result becomes a flat line going through the mean of data. Simply said, Ridge Regression brings regularization to reduce the model complexity by coefficient shrinkage.

$$J(\theta) = MSE(\theta) + \alpha\frac{1}{2}\Sigma_{i=1}^{n}\Theta_i^2 \tag{5.8}$$

Here it is important to note that value of α, the hyperparameter, is not learned automatically by the model; instead, it is set manually and added to the cost function only during training. However, to evaluate the model performance, we use unregularized

performance measures resulting in a different cost function for testing. We fit our model with different values of α over training data to identify the best fit of the model.

Lasso Regression

Least Absolute Shrinkage and Selection Operator (Lasso) Regression is also a regularized Linear Regression. Similar to Ridge, it also adds a regularization term to the cost function. But the regularization term in Lasso is in l_1 norm of the weight vector instead of l_2 norm as in the case of Ridge in Equation 5.8. The Lasso Regression cost function is shown in Equation 5.9.

$$J(\theta) = MSE(\theta) + \alpha\Sigma_{i=1}^{n} \mid \Theta_i \mid \tag{5.9}$$

Just like Ridge, with α equals to zero, Lasso cost function reduces to the cost function of linear regression. Due to the use of l_1 regularization, the coefficients of least important features can lead to zero, i.e. some features are neglected while making predictions. Unlike Ridge, the coefficients can become zero even with a small α value. Due to this, Lasso can provide inherent features selection ability to help reduce the complexity of the model. We fit our model with different values of our hyperparameter and explore the best fit of the model using all the six features as mentioned in the Table 17.

5.2.3 Regression Analysis

We explored the preprocessing of the dataset in subsection 5.2.1. The final dataset after the preprocessing contains 66 entries to be used by the ML models. We apply feature scaling to the dataset and divide it in 3:1 ratio among train and test data respectively. We evaluate Multivariate Linear Regression, Ridge Regression and Lasso Regression over this train and test data exploring different configurations like *cross validation*, and varied hyperparameter values when appropriate.

We start with the Multivariate Linear Regression model and fit the model on train data with and without applying feature scaling over all the features. Without feature scaling, the Table 18 shows the derived intercept and coefficient values of the features for the model. The sign of coefficients of the features appears as expected considering their positive or negative impact on the final measured packet rate. The *Parser* feature coefficient came out with an unexpected value as per common belief. However, during our experiments, and also from the dataset, we have discovered that MacS packet rate does not get influenced a lot when we have greater than 3 headers to be parsed. Hence the small positive coefficient against the *Parser* feature. Considering the coefficient values and the intercept value, we can write our linear model as below:

$$\widehat{y} = 1.697 + 0.973X_1 + 0.120X_2 - 0.574X_3 + 0.043X_4 - 0.665X_5 - 0.091X_6$$

The model performance measures MSE, RMSE, and MAE have values such as 0.576, 0.759, and 0.642 respectively. The R-squared value, the model score, for train and test data are 0.916 and 0.885 respectively which signifies that the model can predict 88.5% of the variability in 'Y' ('Mpps' column).

Table 18 – Coefficient Vector of Linear Regression Model

	Without Feature Scaling	With Feature Scaling
Coefficient Estimates		
Intercept	1.697	2.897
cpus	0.973	2.303
parser	0.120	0.119
parserbranch	-0.574	-0.463
fieldmod	0.043	0.091
exact	-0.665	-0.931
lpm	-0.091	-0.135
Performance Measures		
MAE	0.642	0.642
MSE	0.576	0.576
RMSE	0.759	0.759
R2	0.885	0.885

Next, we repeat the same task after applying feature scaling over the dataset. The Table 18 shows the derived intercept and coefficient values of the features for the linear regression model. With feature scaling, the model performance measures MSE, RMSE & MAE are evaluated to have values such as 0.576, 0.759 & 0.642 respectively. The R-squared value, the model score, for train and test data are 0.916 and 0.885 respectively. These values confirm that feature scaling does not affect the final score and error measurement in the linear regression model, though the coefficients see some variations in their values.

Before going forward, we need to identify if the regression model is well suited for our task or not. Residual analysis is one of the methods to assess if linear regression model is appropriate for the dataset available. We carry out the residual analysis by defining residuals and drawing a residual plot. *Residual* (e) is nothing but the difference between the measured value of the dependent variable (y) and the predicted value ($\widehat{y}$), or in other terms, it is the error in prediction. Each data point will have one residual value. In a dataset, the sum and the mean of the residuals are always equal to zero.

$$Residual(e) = Measured\ Value\ (y) - Predicted\ Value\ (\widehat{y})$$

Residual Plot at Figure 42 is a scatter plot with Residuals in the vertical axis. The horizontal axis representing independent variables intercepts the vertical axis at zero. The objective is to find a pattern in the residual plot. The figure shows a random pattern where

the residuals fall randomly around the horizontal axis. This random pattern indicates linear relations between independent and dependent variables (i.e., features and packet rate) allowing a linear model to fit to the dataset with a good score. With the confidence from the residual plot analysis, we continue exploring linear regression and its regularized variations over our dataset.

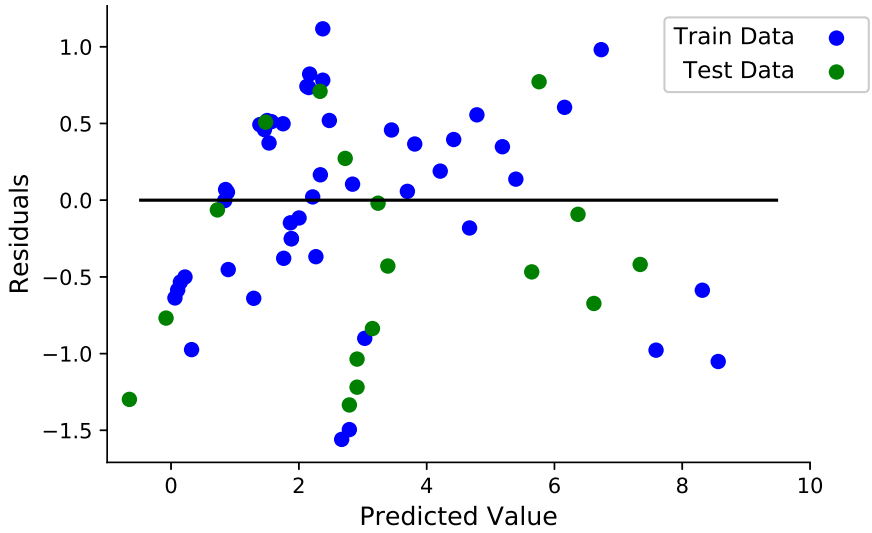


Figure 42 – Residual Plot for Linear Regression

The dataset with feature scaling applied is used to fit the Regularized Regression models, in our case (1) Ridge Regression, (2) Lasso Regression. We have selected an array of values for our hyperparameter α as $[0.0001, 0.001, 0.01, 0.1, 1, 10]$. We fit and test the prediction of the models for every α value to figure out the best fit model. Table 19 shows the R^2 score for the models for all α values over train and test data. We observe that for α value 0.01 or less, our models depict higher accuracy. From the score of Lasso model, we observe that with α value 0.01 or lower, the model does not drop any features to reduce the complexity of the model and utilizes all six features. For α value 0.001, and 0.0001 Lasso model has similar R^2 score, whereas α value at 0.01 it has a lower R^2 score. From this observation, we choose to select α value 0.001 as the best α value for the regularized models. This best α value allows our regression model to keep all the features and to have the best R^2 score too to combat collinearity and overfitting of the models. Note that, for α value as 0.1 and 1 the number of features is reduced to 4 & 1 respectively for Lasso. Moreover, with a higher α '10', our model drops all the features and make the prediction only with the regularization term which is not ideal. In Table 20, we depict the coefficient estimates and different performance measures for the regularized models for our selected best α value '0.001'. We can observe that both Ridge and Lasso regression models have a similar R^2 score performing with similar accuracy. In fact the R^2 score for multivariate linear regression and the regularized models are almost equal as seen in Table 18 & 20.

Table 19 – Ridge Regression Model R^2 Scores

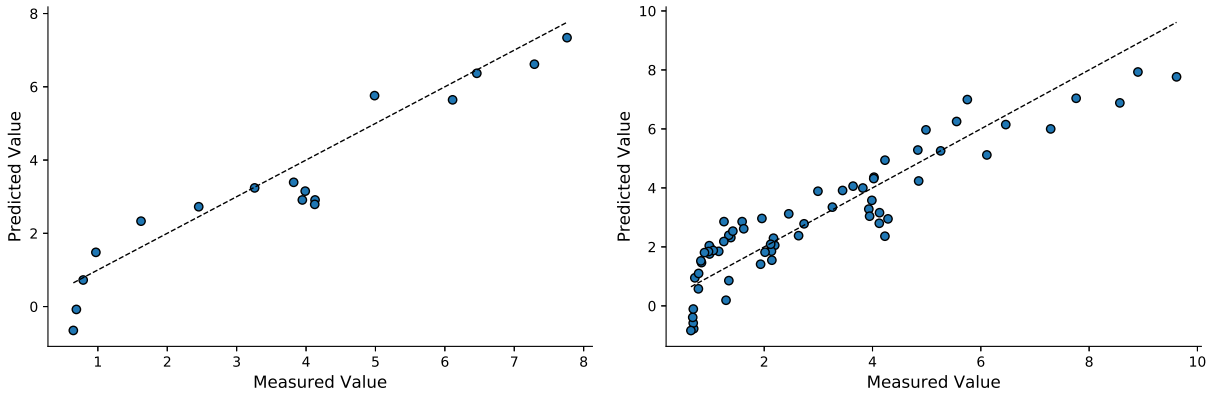
Ridge Regression*			Lasso Regression*		
Train	Test	Aplha (α)	Train	Test	Coeff-Used
0.917	0.885	0.0001	0.917	0.885	6
0.917	0.885	0.001	0.917	0.883	6
0.917	0.885	0.01	0.916	0.858	6
0.917	0.884	0.1	0.905	0.821	4
0.916	0.876	1	0.502	0.273	1
0.864	0.787	10	0	-0.185	0
*R^2 Score					

Table 20 – Regularized Model with α value as 0.001

	Ridge Regression	Lasso Regression
Coefficient Estimates		
Intercept	2.897	2.894
cpus	2.303	2.303
parser	0.119	0.108
parserbranch	-0.462	-0.451
fieldmod	0.091	0.098
exact	-0.930	-0.926
lpm	-0.135	-0.130
Performance Measures		
MAE	0.642	0.649
MSE	0.576	0.587
RMSE	0.759	0.766
R2	0.885	0.883

Finally, we apply *Cross Validation (CV)* to explore how our model behaves with a bigger dataset. We apply 10 K-fold cross-validation taking the dataset size to 660 in total for our model. Figure 43 presents two scatter plots depicting the measured packet rate and predicted packet rate before and after applying cross-validation. This provides a visual representation of how good the prediction of our linear regression model is. The Figure 43b shows far more data points than Figure 43a as a result of the 10 fold cross validation with increased dataset size.

The Table 21 shows the cross prediction values of the performance measures (i.e., MAE, MSE, RMSE, and R^2) for all three of our models. We observed that the R^2 score decreases after applying cross-validation for all our models. Cross-validation helps to reduce the overfitting, but the result may get biased as it reuses the same data while learning in



(a) Without Cross Validation (b) With Cross Validation

Figure 43 – Packet Rate (Predicted Value vs Measured Value).

different folds. Hence the reduction in R^2 score is a possible outcome. Moreover, considering the fact that the R^2 reduction is around 10% at max, and the R^2 values are reasonably high, we deduce that the three linear models performed well on our dataset. And we also believe that the R^2 score over cross-validation can be considered for prediction due to its accuracy. We note that the regularized model performed well in current dataset, and it can continue performing well in future when we will have more data and new features in our dataset as regularization can bring down the complexity of the model and limit the overfitting for the future dataset.

Table 21 – Performance Measures of Different Regression Models

Regression Model	Cross Validation	Performance Measures			
		MAE	MSE	RMSE	R^2
Linear	No	0.642	0.576	0.759	0.885
	Yes	0.839	0.7967	0.892	0.764
Ridge	No	0.642	0.576	0.759	0.885
	Yes	0.764	0.797	0.892	0.839
Lasso	No	0.649	0.587	0.766	0.883
	Yes	0.763	0.797	0.893	0.839
*Aplha 0.001 for Ridge and Lasso					

5.3 Concluding Remarks

This chapter provided a comprehensive overview of the methodology to describe complexity, and presented the complexity of the individual MACSAD use cases in section 5.1 and Table 16 respectively. The chapter proceeded with bringing machine learning

algorithms to create complexity model for MACSAD in section 5.2. We bring the details of data preprocessing and also actually train machine learning models to predict performance value for new inputs while evaluating and comparing three different types of regression models: Multivariate Linear Regression, Ridge Regression & Lasso Regression.

Our machine learning models use features extracted from P4 programs while ignoring other factors such as the target platform parameters or input traffic types etc. We wish to include more features into our dataset and also try to gather more data inputs to keep improving our model. More to it, by introducing an automated complexity feature extractor from P4 program, we are also thinking of using the trained model as a service to predict the performance of any P4 program automatically without any manual effort.

To extend our effort to explore the design of MACSAD, calculate the performance, & evaluate the complexity of its use cases while working as an observer, we now turn our focus on improving the performance by bringing optimization to MACSAD as presented in chapter 6.

6 Optimization

Compiler systems are known for incorporating numerous optimization techniques under the hood: may it language specific or independent; platform specific or independent. Being a multi-target compiler system, it is imperative for MACSAD to focus on similar optimization techniques; and as such memory-level parallelism between CPU and main memory, is integrated into the code auto-generation stage of MACSAD, is detailed here.

6.1 MACSAD Packet Processing Optimization

MACSAD implements various optimization techniques in its different architecture modules. However, our work towards exploiting the memory-level parallelism between CPU and main memory (BHARDWAJ *et al.*, 2017) deserves attention for its impact on performance by targeting the memory-bound steps of packet processing, i.e., the table lookup which consists of table key creation and the actual lookup step. We observed that different lookups share commonality in terms of the basic steps involved in the process. This allows us to implement table key creation and table lookup in batches to exploit the memory level parallelism while hiding the CPU-memory latency. Apart from table lookup, *Parser* is also considered as parsing is performed for every received network packets and similar to tables it is also predefined in a P4 pipeline. By bringing batching to the parser, we can bring parallelism and in turn bring performance improvement via packet rate of MacS. In a switch pipeline, two types of parallelism are found. Data parallelism is the first kind focusing on the ability to simultaneously process different parts of the same packet. Second, the ability to perform different operations on different packets known as pipeline parallelism. We have shown how to exploit pipeline parallelism with multi CPU support over different target platforms and different set of CPU cores. In this section, we focus on data parallelism exploit to improve packet rate of MacS. We explain how MACSAD code is modified to implement batch parsing and batch table lookup to bring data parallelism and related improvements onto MacS. Although batching might introduce more latency in MacS packet processing, the gain in packet rate make it appealing to the pipelines where higher throughput is necessary. With this activity we also exhibit the flexibility aspect of MACSAD which allows designing pipelines focusing on either higher packer rate or lower latency. We bring a clear insight to this activity by observing the experimental results, and analyzing the improvements achieved.

More abstractly, a software switch has ingress components shown in Figure 44 where it receives network packets from RX queues and map the packet structures to the main memory. Followed by, packet processing consists of packet parsing and table

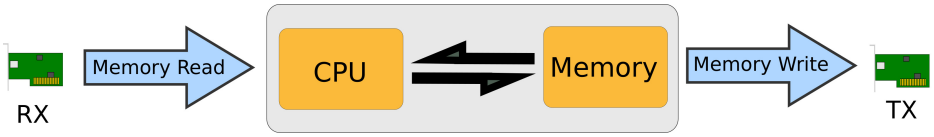


Figure 44 – Abstract Packet Processing Pipeline

lookup, and finally the Egress step where the packet is forwarded by writing back the packet structures to the output TX queues. While the Ingress and Egress components are limited by the underlying NIC capacity, the packet processing node is the component to focus on as this takes the most time to complete and comes under our purview of optimization as well. Modern NICs provide DMA interface to access the NIC Ring buffers directly from userspace. As this NIC-MEMORY DMA interface is a high-bandwidth and high-latency interface, MACS always receives/transmits network packets in batches of burst size from/to the RX/TX NICs. By bringing parallelism in terms of batch of packets receive/transmit, we can amortize the cost of NIC-Memory latency. Being said that, it is possible to assume that MACSAD also process the batch of packets in parallel during the parsing and table lookup activities. However, currently, MACSAD doesn't take advantage of this parallelism to the full extent and internally process one packet at a time per thread. We will focus on this serialized packet processing of MACSAD and impact of parallelism over it.

```
static void
    parse_state_parse_ethernet(
    packet_descriptor_t* pd,
    uint8_t* buf, lookup_table_t**
    tables)
{
  extract_header_ethernet(buf, pd)
  buf += pd->headers[
    header_instance_ethernet].
    length
  return apply_table_smac(pd)
}
```

Listing 6.1 – *Parser (no optimization)*

```
static void parse_state_parse_ethernet(
    packet_descriptor_t* pdt, int
    pkt_cnt, lookup_table_t** tables)
{
  packet_descriptor_t *pd = NULL
  for(int i=0; i < pkt_cnt; i++){
  pd = &pdt[i]
  uint8_t* buf = (uint8_t*) pd->pointer
  extract_header_ethernet(buf, pd)
  buf += pd->headers[
    header_instance_ethernet].length
  apply_table_smac(pd)
  }
}
```

Listing 6.2 – *Parser (with optimization)*

Parsing, table lookup, and table match actions are memory intensive operations with multiple memory access, and cache misses. Towards reducing the footprint of these memory intensive operations, we plan to take advantage of the available CPU-Memory parallelism. Being memory bound our use cases can be beneficial with this CPU-Memory parallelism. The interface between CPU & memory is of high-bandwidth and high-latency as with NIC-MEMORY interface. Modern CPUs can perform multiple memory requests in

parallel. We can take advantage of this memory level parallelism by performing parsing, table lookup, etc., for a burst of packets instead of a single packet at a time. This is implemented as batching, to achieve data locality, in the underlying MACSAD source code so that multiple packets can be processed simultaneously taking advantage of the memory parallelism. This approach of batching is also known as Loop-Fission in the terminology of compiler optimization.

Listing 6.1, 6.2 shows how we bring batching to the MACSAD parser code. We present the code of L2FWD use case where MacS parse only the Ethernet header of a network packet. Listing 6.1 shows that the parser function is called once for each network packet. But Listing 6.2 shows that the parser function takes an additional argument as the number of packets while the *packet_descriptor* argument is modified to point to an array of packets instead of a single packet. This allows to do the parsing for multiple packets using a loop which improve the data locality and improve memory parallelism while reducing the cache misses too.

```
void apply_table_smac(
    packet_descriptor_t* pd,
    lookup_table_t** tables)
{
  uint8_t* key[6];
  uint8_t* value;
  table_smac_key(pd, (uint8_t*)
    key);
  EXTRACT_BYTEBUF(pd,
    field_instance_ethernet_srcAddr
    , key)
  value = exact_lookup(tables[
    TABLE_smac], (uint8_t*)key);
}
```

Listing 6.3 – *Lookup (no optimization)*

```
void apply_table_smac(packet_descriptor_t
    * pdt, int pkt_cnt, lookup_table_t**
    tables)
{
  uint8_t* key[MAX_PKT_BURST][6];
  uint8_t* value[MAX_PKT_BURST];
  packet_descriptor_t* pd = NULL;
  for (int i=0; i < pkt_cnt; i++) {
    pd = &pdt[i];
    EXTRACT_BYTEBUF(pd,
    field_instance_ethernet_srcAddr, key[i
    ])
    value[i] = exact_lookup(tables[
    TABLE_smac], pkt_cnt, (uint8_t*)key[i
    ]);
  }
}
```

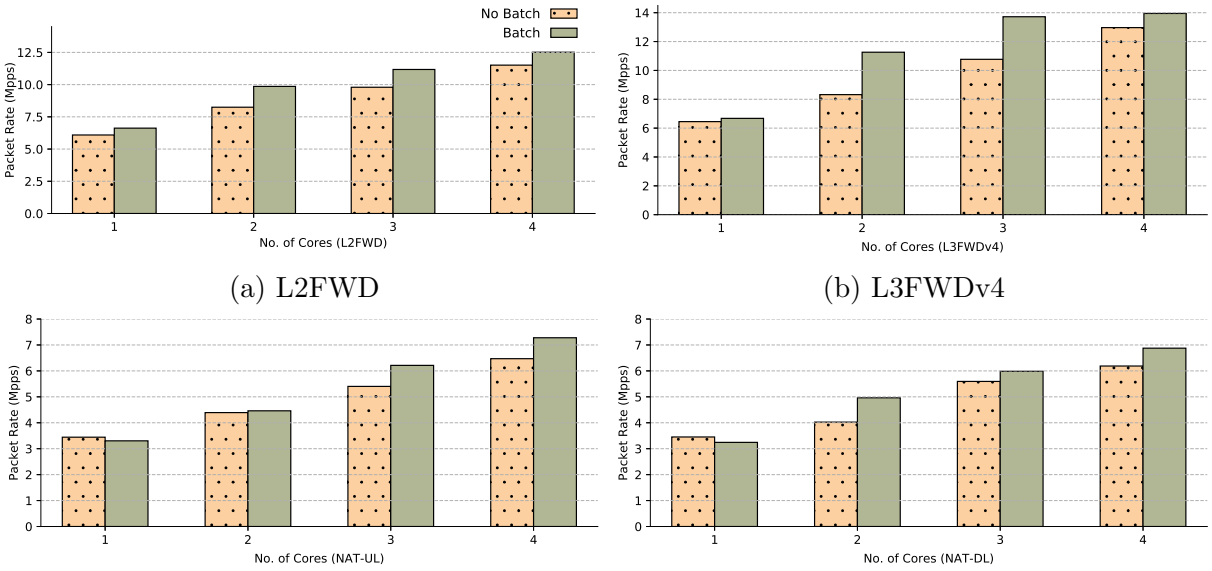
Listing 6.4 – *Lookup (with optimization)*

Listing 6.3, 6.4 presents the MACSAD source code implementing batch concept for table lookup operations for L2FWD use case. L2FWD use case performs P4 Exact lookup based on cuckoo hash over MAC*source* and MAC*dest* addresses for every network packet. Listing 6.3 shows the function for SMAC table of L2FWD use case as appear in the MACSAD source code. This function gets the packet descriptor of the network packet as a function argument for which it creates the Lookup key and performs table lookup. But these operations are done for a single packet each time. Listing 6.4 shows batch approach implemented for the same table related function where it receives packet count as an extra argument while the packet descriptor argument is modified as done

before for parser function in Listing 6.2. Having explained this, we want to point out that our implementation of batching on Table Lookup has its limitation too. We bring batch technique only to the first table in the pipeline while the other tables behave the same as in our current implementation. Nevertheless, the results in Table 22 shows that even with this limited implementation, we achieve improvement in packet rate.

6.2 Evaluation and Analysis

Figure 45 presents the effect of batching on packet rate for different use cases executed with different number of CPU cores. The results for with and without batching optimization are presented in different colors: green & yellow respectively. With the code to exploit memory level optimization MacS performs better with higher packet rate. The increase in packet rate remains effective when we move from single core to multiple CPU core configurations too. Figure 45a shows the improvement in packet rate across all core combinations as expected. We can safely say that the introduction of simple loop fission transformation technique can result in increase of packet rate for MacS across different use cases.



(a) L2FWD (b) L3FWDv4

(c) NAT-UL (d) NAT-DL

Figure 45 – Packet Rate comparison for all Use Cases with batch optimization for different CPU Cores (64 Bytes, 100 Table entries) on Testbed A.

Table 22 depicts a detailed report in terms of percentage gain or loss in packet rate for all the use cases with 1, 2, 3 & 4 CPU core configurations. We perform this experiment in two phases of optimization. The first optimization implements batching for Parser only and refereed as OPT1. The second level of optimization considers batching both for Parsing and Table Lookup, and refereed as OPT2. The table shows the percentage change

in packet rate for both the optimization levels separately for better analysis. We can see that for L2FWD use case the packet rate increases for all the different configurations. But L3FWDv4 use case shows a decrease in packet rate in case of OPT1, whereas for OPT2 the packet rate increases as expected. L3FWDv4 parser looks into 2 headers (Ethernet, IPv4) of the network packet. It also implements a branch instruction while parsing Ethernet header to check if the next header is IPv4 or not. For every branch instruction, MacS actually does a calculation of the header field offset and width, and then performs a *memcopy* to retrieve the data from packet header necessary to execute the conditional instruction. But there is no other statement or operations following up in the parser code block to take advantage of this cache refresh and to amortize the cost, hence the negative impact on the packet rate of L3FWDv4 use case with OPT1. After applying the OPT2 (Table Lookup Optimization), the packet rate improvements are apparent again in L3FWDv4 as shown in the table. Comparing to L2FWD, L3FWDv4 use case shows a definite improvement with OPT2 optimization. This leads us to believe that the LPM lookup method for IPv4 get more advantage compared to CuckooHash method for MAC address lookup under this activity.

Table 22 – Use Case Performance Results (64 Bytes, 100 Table Entries, DPDK, Testbed A)

	L2FWD		**L3FWDv4**		**NAT-UL**		**NAT-DL**		
	Optimization Type								
	OPT1	**OPT2**	**OPT1**	**OPT2**	**OPT1**	**OPT2**	**OPT1**	**OPT2**	OPT1 (Parser)
No of CPUs	% Increase in Packet Rate								
1	8.09	8.72	-13.52	3.50	-3.65	-4.08	-2.88	-6.03	OPT2 (Parser + Lookup)
2	19.02	19.58	-6.20	35.34	5.37	1.58	7.22	23.31	
3	12.23	14.08	-3.51	27.41	5.67	15.01	-0.17	6.99	
4	7.16	8.70	-3.33	7.57	4.63	12.48	3.79	11.13	
	L2FWD		**L3FWDv4**		**NAT-UL**		**NAT-DL**		

Looking at the results for NAT use case we note that both NAT-UL and NAT-DL show a definite improvement in packet rate for multi-core configurations while a reduction in packet rate for single core configuration. Parser optimization (OPT1) brings improvements to NAT use case, unlike the L3FWDv4 use case. NAT-DL parser code in Listing 6.5 shows that NAT-DL performs parsing for 3 packet header namely Ethernet, IPv4 and TCP headers. Both Ethernet and IPv4 header parsing code has a branch condition similar to L3FWDv4 use case to identify the next available header in the network packet. But NAT-DL shows an increment in packet rate with OPT1, unlike L3FWDv4 which shows a decrease in packet rate. For NAT-DL the Ethernet parsing does a cache refresh and prefetch additional nearby data which corresponds to the next header details. Hence the following header parsing takes advantage of this warm cache and brings down the average cost for parsing. Due to this, NAT-DL performs better with OPT1 compared to L3FWDv4 although both have similar code in their parser modules.

```
static void parse_state_parse_ethernet(packet_descriptor_t* pd,
    lookup_table_t** tables)
{
  uint8_t* buf = (uint8_t*) pd->pointer;
  extract_header_ethernet(buf, pd);
  build_key_parse_ethernet(pd, buf, key);
  uint8_t case_value_0[2] = {8, 0, };
  if ( memcmp(key, case_value_0, 2) == 0)
    parse_state_parse_ipv4(pd, buf, tables);
}

static void parse_state_parse_ipv4(packet_descriptor_t* pd, uint8_t* buf,
    lookup_table_t** tables)
{
  extract_header_ipv4(buf, pd);
  EXTRACT_INT32_AUTO(pd, field_instance_ipv4_srcAddr, value32)
  pd->fields.field_instance_ipv4_srcAddr = value32;
  pd->fields.attr_field_instance_ipv4_srcAddr = 0;
  EXTRACT_INT32_AUTO(pd, field_instance_ipv4_dstAddr, value32)
  pd->fields.field_instance_ipv4_dstAddr = value32;
  pd->fields.attr_field_instance_ipv4_dstAddr = 0;
  build_key_parse_ipv4(pd, buf, key);
  uint8_t case_value_0[1] = {6, };
  if ( memcmp(key, case_value_0, 1) == 0)
    parse_state_parse_tcp(pd, buf, tables);
}

static void parse_state_parse_tcp(packet_descriptor_t* pd, uint8_t* buf,
    lookup_table_t** tables)
{
  extract_header_tcp(buf, pd);
}
```

Listing 6.5 – *NAT-DL Parser with out Optimization*

For both Parser & Table Lookup Optimization (OPT2), NAT use cases show an additional increase in packet rate for both NAT-DL and NAT-UL for multi-cpu configurations. In the case of a single core, there is a decrease in packet rate similar to OPT1. This can be explained by the way batching is implemented in MACSAD for Table Lookup.

MACSAD autogenerates code for table *match+action* abstractions from P4 program. In P4, we can define tables with or without conditional statements to dictate the sequence, and in turn setting the depth of the pipeline. This allows supporting multiple pipelines with different sets of tables in the same P4 program. Although in P4 the conditional statement to decide next table is optional and the tables are selected in a sequence as present in the P4 code, in MacS we follow a different approach to facilitate the auto-

generation of code. MACSAD employs a mandatory conditional statement to specify the jump to the next table. And in the absence of such a branch condition, it uses a default condition to go to the next table. Hence in MACSAD, the tables are always dependent on the previous table. We implement batching for Table lookup only for the first table in the pipeline. By adding batch technique in table lookup for the first table, we get improved packet rate because of the memory locality and cache hit as packet processing continues over the packet header details already prefetched into the cache. This behavior is limited to memory bound use cases only. In case of a CPU bound situation, the optimization can even become harmful by exerting pressure on the cache system which is apparent in the result for the single core configuration tests. When we run the same experiment with multiple cores, the underlying compiler is able to identify the loop fission and bring instruction level parallelism into effect. As a result, the compiled code can utilize SIMD and instruction level parallelism to increase the packet rate in multi-core configurations as seen in Table 22.

NAT use cases have 5 tables in the pipeline where the first table "if_info" identifies the traffic as Uplink or Downlink. In our testbed with only two ports, we have one table entry in this table for traffic type identification. With only one entry, the compute boundness of table lookup is more towards CPU bound than memory bound. The performance dips are due to increased pressure on the caching subsystem with increased memory footprint due to the batch size. It is important to note here that batching implementation are ineffective or has a negative impact on compute bound operations as explained before. While moving on to the multicore configuration, we find an increase in packet rate as the batch features implemented as loop brings memory and data locality dictating the compiler to take advantage of SIMD and instruction level parallelism as explained already.

6.3 Concluding Remarks

In this chapter, we elaborated the idea of bringing optimization into MACSAD as introduced in subsection 1.3.4 with a goal to observe performance improvement. We explained and implemented optimization techniques to leverage memory-CPU parallelism. This activity brings changes to the *Transpiler* to auto-generate codes expressive enough for the low-level compiler to create optimized compiled output. Loop-Fission optimization technique is explored extensively for this activity bringing the optimization in two-fold for packet parsing and table lookup part of the MACSAD data plane. The details of optimization techniques and the supporting performance results are shown in section 6.1 and section 6.2 respectively.

With this chapter, we wrap the discussion encompassing different aspects of MACSAD, and bring our attention from MACSAD itself to the multiple contributions done

as part of the MACSAD development and also as part of other open source project development. All the artifacts comprising of the new open-source tool, and code contribution as feature addition to existing open-source projects are presented in chapter 7.

7 Open Source Artifacts

MACSAD is an open source project built upon a number of different open source and free softwares to achieve its goal. In the process of developing MACSAD, we encountered a number of challenges due to the shortcomings of the participating libraries and softwares, and also due to the lack of any open source alternatives for our requirements. We diligently kept record of every major bugs or defects encountered during our research and prototyping of MACSAD. We dutifully notified the issues to corresponding development team via email or mailing list as appropriate. In the case of lack of an alternative free/open source software, we took the initiative to create a tool to fulfill our requirement. Other instances when we found that contributing to an existing open source software as a new feature was necessary, we delivered by submitting our code too. All these additional efforts are our way of giving back to the community without whose help we could not have completed our prototyping and release of MACSAD. The following sections provide a complete view of these contributions.

7.1 BB-Gen Tool

While prototyping MACSAD, we faced challenges to find traffic traces to verify our use cases. The readily available network traffic traces come in PCAP format. In a traditional workflow, we try to search for a PCAP file as per our need, and then we read the PCAP file to verify the traffic flow type available in it. Then we read the network packet headers to get the details like MAC*source*, IP*source*, TCP*port*, etc. These details then put in a CSV file in a specific format to match the tables in the use case pipeline. These CSV files are simple text files and also known as *Table Trace File* as they are used to update the use case pipeline tables. The current tools available to work on PCAP files are slow, clumsy and non-intuitive. We also need to put extra effort to learn those tools and write our own scripts to use the output of those tools to create our Table Trace file. For a PCAP file with a million flows, these process can take a day without accounting the unforced error occurs due to the unavailability of details about the PCAP file. Similarly, for any small change in use case pipeline, the whole process needs to be redone which frequently happens during prototyping of a project. Similarly, we need to undergo through these processes every time a new use case is added. Hence we were in the lookout for a tool to provide Table Trace files in a small amount of time for all our use cases without the huge additional effort for any future modifications in the use cases.

When we choose to select PCAP files from any open repositories, we are forced to use the flow distribution available in those PCAP files. In case we want to create our own

variance in flow distribution, we again need to depend on the PCAP editing tools which are cumbersome to work with. Hence we wanted a tool which can create PCAP and Table Trace files based on our required flow distribution.

MACSAD has support for a number of use cases varying the pipeline complexity while working with different tunneling protocols. Hence we wanted a tool which can provide PCAP and Table Trace fiels for all the use cases using the exactly similar flow distribution so that the comparison among the use cases can be adequately performed as the effect of flow distribution of network traffic across the use cases remain similar.

While working with P4, we saw an opportunity in terms of a packet crafter tool which can parse a P4 program and create a workload trace (PCAP) for the defined pipeline. This P4 dependent workload trace can help to automate any effort towards testing of P4 defined switches, MacS in our case. A P4 based packet crafter tool is identified as a need of the hour considering the proliferation of P4 language in the research community.

To sum it up we identified the following requirements for a packet crafter tool and decided to go ahead with creating our own referred to as BB-Gen.

- Can create workload traffic as PCAP file from the P4 pipeline.
- Ability to generate both PCAP and Table Trace file to facilitate testing of network switches.
- Can create PCAP and Table Trace files simultaneously in a small amount of time for different use cases to maintain the same flow distribution across the use cases.
- Ability to create PCAP files for different packet sizes as per RFC 2544 (BRADNER; MCQUAID, 1999).
- Easy to add support for new protocol headers & tunneling protocols.
- Command Line Interface (CLI) with intuitive and easy to use options.
- Written in a popular language, Python, so that it will be easy to maintain and contribute.
- Support for randomized header field data generation for worst case testing scenario.

Our solution BB-Gen identifies itself as a simple CLI based packet crafter written in Python language. It implements the Scapy library for all the PCAP related activities. It is modular, and separates the logic to create the flow distribution details and actual packet & PCAP creation process. This allows us to first create the header details for all the protocols only once, and then use the same details to create traffic as PCAP files for

different packet sizes and for all the use cases supported, and the corresponding Table Trace fields. In addition, BB-Gen has also been expanded to work with other switches like T4P4S for creating PCAP and Table Trace files.

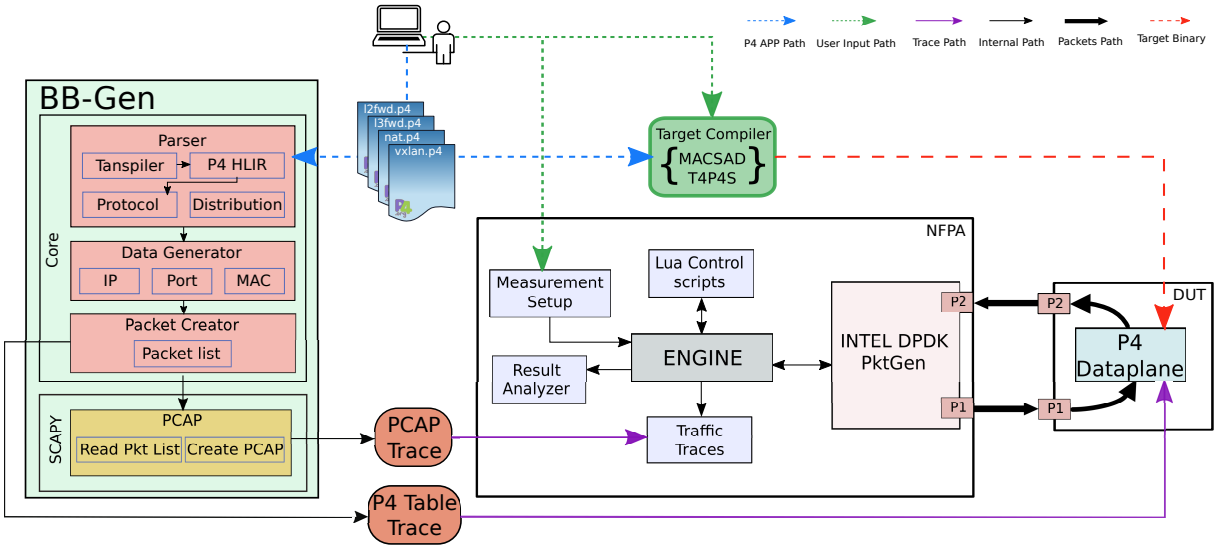


Figure 46 – BB-Gen Architecture and Integration with NFPA and MACSAD & T4P4S

Figure 46 shows BB-Gen architecture and also its integration with different Switch and Traffic generators in a traditional test topology. For MACSAD too we employed this type of workflow as explained in section 4.1. The left side of the figure shows the architecture of BB-Gen. The flow and flow distribution details generation, and the actual PCAP creation are implemented using two different modules as *Core* and *Scapy* respectively. The Core module workflow involves three submodules and in turn 3 steps: Parser handles P4 file to extract header details and to create corresponding data structures in BB-Gen for header data generation; *Data Generator* actually creates the header fields and protocol data based on the user input flow distribution logic; Packet Creator assembles all the header fields and create payload for different packet sizes (i.e., from 64 to 1500 Bytes), and generate the Table Trace files from these details. Then *PCAP* submodule assembles the PCAP files using SCAPY library.

As per the diagram, the PCAP files are meant to be used by traffic generators like NFPA (CSIKOR *et al.*, 2015) while the Table Trace files are inputs to the MACSAD or other supported switches (e.g., T4P4S) to update the pipeline tables in the switch. This completes the cycle of a basic unit testing for any switch pipeline, in our case MACSAD. We used BB-Gen extensively to create the network traffic or PCAP files and the corresponding Table Trace files during our testing and evaluation of MACS explored in section 4.3, section 4.4, and section 4.5. The BB-Gen tool has been open-sourced and available on GitHub [1] for the research community to take advantage of it.

[1] https://github.com/intrig-unicamp/BB-Gen

7.2 OpenDataPlane (ODP)

7.2.1 Issues and Fixes for ODP

OpenDataPlane (ODP) is an integral part of MACSAD architecture. While prototyping MACSAD and doing testing & evaluation we encountered some obstacles. Further analysis were done each time to identify the root cause of the issues related to ODP. Due to diverse categories of our testing scenarios, we are able to unearth multiple issues with ODP which went unnoticed before even by the vast variety of automated unit tests already in place for ODP. The issues varied from LPM lookup method to DPDK driver related implementation. All the issues are promptly reported to the ODP community via mailing list, and we followed up with providing more details about the issue, to test the fixes on our test bed before making it live and providing unit test scripts to recreate the issue by other developers.

- While testing our L3FWDv4 use case, we figure out that the LPM implementation in place didn't work well when both KEY & VALUE is stored in the table itself. LPM table was capable of storing an index or an integer as the VALUE of the KEY VALUE pair of a table lookup. We help to identify the issue followed by helping to validate the patch provided by the ODP developers for the same.
  <https://github.com/Linaro/odp/pull/701>

- In our effort to test MacS over different testbed can sometime bring surprises. During one of these experiments, we found that ODP throw error message *"Segmented buffers not supported"* with DPDK packet I/O for larger packet sizes (1518 Bytes). We identified that some DPDK NICs need at least 2176 byte buffers (2048B + headroom) not to segment standard Ethernet frames. We worked with ODP developers to create a patch for the DPDK driver in ODP with increased minimum segment length to avoid this issue.
  <https://github.com/Linaro/odp/pull/731>

- During our MacS test at **Testbed B**, we encountered a problem with ODP not working with DPDK Packet I/O with our Mellanox 100G NICs. We were able to figure out the root cause for this to be unbalanced hugepage memory allocation for NUMA nodes. After reporting this, we continuously provided more details about the issue and also helped to validate the fix for this bug.
  <https://bugs.linaro.org/show_bug.cgi?id=3657>

- For our adaptive dynamic CPU core allocation experiment, we worked on two different Testbed with 3 different types of NICs. Due to these unique configurations, we unearthed another issue with ODP. We found that some Intel NICs (e.g., 82599,

X540) stop receiving packets from all RX queues of the NIC if any RX queue is not emptied fast enough or if any configured RX queues remain unused. Due to the nobility of this use case, this was never tested for ODP before. We worked closely with developers and provided our Testbed for testing and verifying the fix which was essential for our MACS evaluation.
<https://bugs.linaro.org/show_bug.cgi?id=3618>

7.2.2 IPv6 support for LPM Lookup in ODP

ODP code base does not provide IPv6 support for LPM lookup table implementation, although ODP has limited support for IPv6 protocol support. To fulfill L3FWDv6 use case implementation, we extend the current IPv4 based LPM lookup to bring IPv6 support. Our IPv6 implementation is based on the IPv4 based LPM lookup library. We extend the IPv4 Binary prefix tree to support 128 bits key or addresses towards IPv6 protocol support. We also create a complete library for IPv6 LPM which includes corresponding table management APIs for table creating, table entry addition and table entry deletion. The source code for IPv6 support for LPM lookup can be found here.
<https://github.com/ecwolf/odp/tree/ipv6/helper>

7.2.3 Contribution for odp-thunderx

Towards portability, we always try to bring MACSAD over different target platforms. As mentioned in section 4.1 we have brought MACSAD to Cavium with ThunderX & OCTEON chipsets by using the ODP version provided by Cavium. This ODP for Cavium (odp-thunderx) is based on an older version of ODP missing CuckooHash and LPM lookup implementations. We contributed CuckooHash and LPM lookup related code to *odp-thunderx* code base. For completeness, we also implemented IPv6 support for LPM lookup method. Further, we added *crc32c* hash for improvements to CuckooHash implementation. All the contribution towards CuckooHash, LPM with IPv4 & IPv6 support and *crc32c* hash support are available in GitHub here.
<https://github.com/c3m3gyanesh/odp-thunderx>

7.3 Additional Open-source Contributions

In the course of MACSAD development, we have made multiple contributions towards P4, NFPA and other projects in addition to the contributions explained before. These contributions are shown here.

- We contributed to p4c-graphs code base which generates dependency graphs from a P4 program file. The p4c-graphs tool was only able to create the dependency

graph for control blocks in case of $P4_{16}$ program whereas the ability to generate graphs for parser was missing. We created a feature request, contributed code as a pull request to the GitHub repository and followed up with another contribution towards another issue raised for p4c-graphs tool too. With this contribution, p4c-graphs can generate the dependency graph for top-level parser blocks and present it as a dot file. This contribution is also responsible for generating all the parser dependency graphs used in this text.
<https://github.com/p4lang/p4c/pull/969>
<https://github.com/p4lang/p4c/issues/1038>

- Our contribution for NFPA is a simple feature enhancement bringing additional configuration parameter to NFPA. With our contribution, it is possible to set the maximum line rate for an interface under NFPA so that the packet transmit rate is controlled to a specific percentage of the theoretical maximum of the corresponding NIC. Our code contribution is available here.
<https://github.com/cslev/nfpa/pull/2>

- We have a Github public repository for all our use cases. One can find a simple description and pictorial representation of the use cases and their scenarios. The P4 program files in both $P4_{14}$ & $P4_{16}$ format for all the use cases are also included. And finally, the Parser & Table dependency graphs are also provided in this repository. This public repository is available here.
<https://github.com/intrig-unicamp/macsad-usecases>

- Support for P4 language syntax highlighting were missing in modern text editors. To help us with our P4 programming, we created a syntax highlighter for P4 for the Sublime Text editor. We also created a P4 syntax highlighter collection bringing together the P4 syntax highlighter for Sublime Text, VIM and EMACS text editors under one repository. Continuing our effort, we contributed to P4 syntax highlighter project for Atom text editor with multiple commits. These efforts are helpful for the programmers to write P4 programs efficiently. The public repositories created and code contributions made are available here.
<https://github.com/c3m3gyanesh/p4-syntax-highlighter>
<https://github.com/c3m3gyanesh/p4-syntax-highlighter-collection>
<https://github.com/Yi-Tseng/atom-language-p4/pull/1>
<https://github.com/Yi-Tseng/atom-language-p4/pull/3>
<https://github.com/Yi-Tseng/atom-language-p4/pull/4>

7.4 Concluding Remarks

This thesis proposal 'MACSAD' is built upon P4, ODP, and other open source projects. During the MACSAD development, we have identified and resolved limitations, issues, and missing features of the contributing open source projects. This chapter explores these contributions starting with our new open-source packet crafter tool BB-Gen in section 7.1. We also explored and presented the IPv6 based LPM lookup and bug fixes for ODP in section 7.2. Finally, we wrapped this chapter with details about new feature addition to P4 and NFPA, and contributions to modern text editors to support P4 syntax highlighting in section 7.3.

8 Future Works & Conclusions

We here discuss the limitations and respective solutions of different aspects of this thesis presented under *Future Works* section. Followed by, we present a brief take on all the aspects of MACSAD and its more significant impact on networking in its completeness to conclude the thesis.

8.1 Future Works

MACSAD represents a promising approach towards portability of data plane applications by transparently compiling the high-level P4 language over to different target platform using just enough low-level platform-independent ODP APIs. We have shown multiple aspects of MACSAD development from design & development to performance optimization, use case evaluation to resource scalability, and from developing new open source tools to contributing code to existing open source projects. Each aspect accomplished some of our goals while bringing new challenges and thought points to become the next new goals a.k.a., Future Works. We delve into the list of future works which are current limitations of our work while also including those which appeared as new requirements and challenges to be addressed. We present these ideas under three broad categories.

MACSAD Design Related.
We begin our discussion with MACSAD design limitations and improvements important to be part of our development roadmap. Looking in bottom-up manner, we start from the target hardware and its architecture, and move upwards till north-bound interface towards SDN controllers. We demonstrated $P4_{16}$ support with BNG use case in MACSAD, but the support is restricted to PISA architecture only. We plan to bring support for the newer PSA pipeline architecture which inherently brings support to PISA and also many more. This allows us to expand the supported target platforms where the targets cannot be mapped to the PISA model limiting MACSAD implementation over them. Following on, we also try to expand the list of $P4_{16}$ feature support by implementing 'Stateful $P4_{16}$ Constructs' in MACSAD. Then finally looking at the interface towards the controller, we are working towards P4 Runtime (PRT) support for MACSAD. With PRT we move closer to achieve programmability in every level of an SDN network device, i.e., data plane, control plane, and management plane.

Towards Virtualization.
With an intent to increase the footprint of MACSAD in virtualization, we intend to broaden our MACSAD as VNF activity. This task is planned in two stages for MAC-

SAD. First, exploring MACSAD in a different virtualized environment such as docker container and virtual machine is of great importance. We plan to bring all the currently supported use cases and evaluate extensively with different configuration and resource parameters in the virtual environment. Second, we extend the adaptive CPU scaling approach, and also include more resource parameters such as memory, CPU frequency, hugepage, etc., possible from the competence from the first step working with different virtualized environment. Meanwhile, the current Adaptive CPU scaling approach is explored potentially in combination with SDN controller feedback loops and core utilization measurements to develop new run-time core allocation algorithms and automated this by adapting the MACSAD design. Finally, in a bigger note, we can bring the findings of these two stages together to present an SDN network where the SDN nodes are MACSAD as VNFs with resources assigned to them are identified and calculated by the SDN controller itself.

Extending Machine Learning Analysis.
Continuing our analysis of MACSAD throughput using machine learning, we want to solidify our approach by bringing improvement to our model. We want to increase the number of features adding the excluded complexity factors from the Table 16. Then we want to work with higher throughput NICs unlike the current 10G one to be able to add more entries into our data set. This helps with our understanding of resource scalability. Apart from these complexity factors based on the data plane application a.k.a. P4 program, we want to extend the feature set to include different configuration parameters of the target device too. These features include CPU frequency, memory available, cache structure, etc., bringing more robustness to our models contra different platforms. With this, we can take our model from Cavium platform to other platforms easily. Currently, the complexity factors are extracted from the P4 program manually. We plan to adopt an automated way for this activity and include this as a part of the steps of our machine learning approach. With this, we plan to bring forth the already trained model as a service where it can predict the performance seamlessly for any input P4 program by first extracting the complexity factors and then applying the model over the extracted features.

8.2 Conclusions

SDN networking is considered as a strong contender bringing flexibility to network and network devices, and introduces programmability to control plane and data plane. In relation to this thesis, as an amalgam of the protocol-independent P4 programming language and the target-independent ODP SDK (OPENDATAPLANE, 2018), MACSAD offers a compiler system towards flexible data plane attaining programmability, portability, performance & scalability (3PS). We explored the design of MACSAD followed

by evaluation of different use cases across platforms. We present MACSAD as VNF exploring how MACSAD can be used in virtualized environment too. We also evaluate resource scalability with adaptive CPU allocation in MACSAD which is extremely useful both in NFV and data centers. We also showed that different optimization methods can be introduced with little changes in MACSAD internals which confirms the design flexibility MACSAD presents. Apart from the different aspect of flexibility, we also introduced predictive benchmarking analysis, less explored area of programmable data plane landscape, to MACSAD. We present our approach to identify various complexity factors and use them as features to create machine learning model for MACSAD. These models are capable of predicting performance values for MACSAD without the need to compile and run a use case. Besides, we have more contributions which include the development of an open-source tool, feature addition and support to open-source projects and many more. The work and results presented in this thesis would contribute as the differentiating factor for the research community helping them to take informed and empirical decisions in contrivance towards flexibility and in turn SDN networking.

Bibliography

ANTICHI, G.; SHAHBAZ, M.; GENG, Y.; ZILBERMAN, N.; COVINGTON, A.; BRUYERE, M.; MCKEOWN, N.; FEAMSTER, N.; FELDERMAN, B.; BLOTT, M.; MOORE, A. W.; OWEZARSKI, P. Osnt: open source network tester. *IEEE Network*, v. 28, n. 5, p. 6–12, Sep. 2014. ISSN 0890-8044. Cited on page 75.

BHARDWAJ, A.; SHREE, A.; REDDY, V. B.; BANSAL, S. A preliminary performance model for optimizing software packet processing pipelines. In: *Proceedings of the 8th Asia-Pacific Workshop on Systems*. New York, NY, USA: ACM, 2017. (APSys '17), p. 26:1–26:7. ISBN 978-1-4503-5197-3. Available from Internet: <http://doi.acm.org/10.1145/3124680.3124747>. Cited 2 times on pages 28 and 131.

BOSSHART, P.; GIBB, G.; KIM, H.-S.; VARGHESE, G.; MCKEOWN, N.; IZZARD, M.; MUJICA, F.; HOROWITZ, M. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. New York, NY, USA: ACM, 2013. (SIGCOMM '13), p. 99–110. ISBN 978-1-4503-2056-6. Available from Internet: <http://doi.acm.org/10.1145/2486001.2486011>. Cited 3 times on pages 23, 32, and 37.

BRADNER, S.; MCQUAID, J. *Benchmarking Methodology for Network Interconnect Devices*. 1999. RFC 2544. Cited 3 times on pages 75, 77, and 140.

BUDIU, M.; DODD, C. The p416 programming language. *SIGOPS Oper. Syst. Rev.*, ACM, New York, NY, USA, v. 51, n. 1, p. 5–14, set. 2017. ISSN 0163-5980. Available from Internet: <http://doi.acm.org/10.1145/3139645.3139648>. Cited 2 times on pages 39 and 41.

CESEN, F. R.; PATRA, P. G. K.; ROTHENBERG, C. E. Bb-gen: A packet crafter for data plane evaluation. In: *SBRC 2018 - Salão de Ferramentas*. [s.n.], 2018. Available from Internet: <http://www.sbrc2018.ufscar.br/wp-content/uploads/2018/04/180625_1.pdf>. Cited on page 26.

CESEN, F. R.; PATRA, P. G. K.; ROTHENBERG, C. E.; PONGRACZ, G. Design, implementation and evaluation of ipv4/ipv6 longest prefix match support in p4 dataplanes. *Workshop em Desempenho de Sistemas Computacionais e de Comunicação (WPerformance_CSBC)*, v. 17, n. 1/2018, 2018. ISSN 2595-6167. Available from Internet: <http://portaldeconteudo.sbc.org.br/index.php/wperformance/article/view/3319>. Cited 2 times on pages 27 and 30.

CHOLE, S.; FINGERHUT, A.; MA, S.; SIVARAMAN, A.; VARGAFTIK, S.; BERGER, A.; MENDELSON, G.; ALIZADEH, M.; CHUANG, S.-T.; KESLASSY, I.; ORDA, A.; EDSALL, T. drmt: Disaggregated programmable switching. In: *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 1–14. ISBN 978-1-4503-4653-5. Available from Internet: <http://doi.acm.org/10.1145/3098822.3098823>. Cited 3 times on pages 23, 32, and 37.

CLARK, D. The design philosophy of the DARPA internet protocols. *ACM SIGCOMM Comput. Commun. Rev.*, v. 18, n. 4, p. 106–114, aug 1988. ISSN 01464833. Available from Internet: <http://portal.acm.org/citation.cfm?doid=52325.52336>. Cited on page 21.

CSIKOR, L.; SZALAY, M.; SONKOLY, B.; TOKA, L. NFPA: Network function performance analyzer. In: *IEEE NFV-SDN*. [S.l.: s.n.], 2015. p. 17–19. Cited 3 times on pages 75, 76, and 141.

D. Zhou et al. Scalable, high performance ethernet forwarding with cuckooswitch. In: *Proceedings of ACM CoNEXT '13*. [S.l.: s.n.], 2013. ISBN 978-1-4503-2101-3. Cited on page 47.

DANG, H. T.; WANG, H.; JEPSEN, T.; BREBNER, G.; KIM, C.; REXFORD, J.; SOULÉ, R.; WEATHERSPOON, H. Whippersnapper: A p4 language benchmark suite. In: *Proceedings of the Symposium on SDN Research*. New York, NY, USA: ACM, 2017. (SOSR '17), p. 95–101. ISBN 978-1-4503-4947-5. Available from Internet: <http://doi.acm.org/10.1145/3050220.3050231>. Cited 2 times on pages 28 and 114.

DIETZ, T.; BIFULCO, R.; MANCO, F.; MARTINS, J.; KOLBE, H. J.; HUICI, F. Enhancing the bras through virtualization. In: *Proceedings of the 2015 1st IEEE NetSoft*. [S.l.: s.n.], 2015. p. 1–5. Cited on page 87.

DPDK: Data Plane Development Kit. 2010. Accessed: 2018-09-24. Available from Internet: <http://dpdk.org>. Cited on page 34.

EGEVANG, K. B.; FRANCIS, P. *The IP Network Address Translator (NAT)*. [S.l.], 1994. <http://www.rfc-editor.org/rfc/rfc1631.txt>. Available from Internet: <http://www.rfc-editor.org/rfc/rfc1631.txt>. Cited on page 82.

ETHERNET switch device driver model. 2014. Accessed: 2018-09-24. Available from Internet: <https://www.kernel.org/doc/Documentation/networking/switchdev.txt>. Cited on page 46.

FARINACCI, D.; LI, T.; HANKS, S.; MEYER, D.; TRAINA, P. *Generic Routing Encapsulation (GRE)*. [S.l.], 2000. <http://www.rfc-editor.org/rfc/rfc2784.txt>. Available from Internet: <http://www.rfc-editor.org/rfc/rfc2784.txt>. Cited on page 87.

FARINACCI, D.; LI, T.; HANKS, S.; MEYER, D.; TRAINA, P. *Generic Routing Encapsulation (GRE)*. [S.l.], 2000. Cited on page 90.

FOSTER, N.; HARRISON, R.; FREEDMAN, M. J.; MONSANTO, C.; REXFORD, J.; STORY, A.; WALKER, D. Frenetic: A network programming language. In: ACM. *ACM Sigplan Notices*. [S.l.], 2011. v. 46, n. 9, p. 279–291. Cited on page 32.

GALLENMüLLER, S.; EMMERICH, P.; WOHLFART, F.; RAUMER, D.; CARLE, G. Comparison of frameworks for high-performance packet io. In: *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. [S.l.: s.n.], 2015. p. 29–38. Cited 2 times on pages 91 and 93.

GREEN, S. B. How many subjects does it take to do a regression analysis. *Multivariate behavioral research*, Taylor & Francis, v. 26, n. 3, p. 499–510, 1991. Cited on page 120.

HAN, S.; JANG, K.; PARK, K.; MOON, S. Packetshader: A gpu-accelerated software router. In: *Proceedings of the ACM SIGCOMM 2010 Conference*. New York, NY, USA: ACM, 2010. (SIGCOMM '10), p. 195–206. ISBN 978-1-4503-0201-2. Available from Internet: <http://doi.acm.org/10.1145/1851182.1851207>. Cited on page 48.

HARRIS, R. J. *A primer of multivariate statistics*. [S.l.]: Psychology Press, 2001. Cited on page 120.

KAWASHIMA, R.; NAKAYAMA, H.; HAYASHI, T.; MATSUO, H. Evaluation of forwarding efficiency in nfv-nodes toward predictable service chain performance. *IEEE Transactions on Network and Service Management*, v. 14, n. 4, p. 920–933, Dec 2017. ISSN 1932-4537. Cited on page 100.

KOHLER, E.; MORRIS, R.; CHEN, B.; JANNOTTI, J.; KAASHOEK, M. F. The click modular router. *ACM Trans. Comput. Syst.*, ACM, New York, NY, USA, v. 18, n. 3, p. 263–297, ago. 2000. ISSN 0734-2071. Available from Internet: <http://doi.acm.org/10.1145/354871.354874>. Cited on page 47.

KREUTZ, D.; RAMOS, F. M. V.; VERíSSIMO, P. E.; ROTHENBERG, C. E.; AZODOLMOLKY, S.; UHLIG, S. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, v. 103, n. 1, p. 14–76, Jan 2015. ISSN 0018-9219. Cited 2 times on pages 22 and 32.

LAKI, S.; HORPáCSI, D.; VöRöS, P.; KITLEI, R.; LESKó, D.; TEJFEL, M. High speed packet forwarding compiled from protocol independent data plane specifications. In: *Proceedings of the 2016 ACM SIGCOMM Conference*. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 629–630. ISBN 978-1-4503-4193-6. Available from Internet: <http://doi.acm.org/10.1145/2934872.2959080>. Cited 2 times on pages 27 and 48.

M. Dobrescu et al. Routebricks: exploiting parallelism to scale software routers. In: *ACM SIGOPS*. [S.l.: s.n.], 2009. Cited on page 48.

M. Shahbaz et al. The case for an intermediate representation for programmable data planes. In: *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*. New York, NY, USA: ACM, 2015. (SOSR '15), p. 3:1–3:6. ISBN 978-1-4503-3451-8. Available from Internet: <http://doi.acm.org/10.1145/2774993.2775000>. Cited on page 39.

M. Shahbaz et al. PISCES: A Programmable, Protocol-Independent Software Switch. In: *ACM SIGCOMM*. [S.l.: s.n.], 2016. ISBN 978-1-4503-4193-6. Cited on page 48.

MAHALINGAM, M.; DUTT, D.; DUDA, K.; AGARWAL, P.; KREEGER, L.; SRIDHAR, T.; BURSELL, M.; WRIGHT, C. *Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks*. [S.l.], 2014. <http://www.rfc-editor.org/rfc/rfc7348.txt>. Available from Internet: <http://www.rfc-editor.org/rfc/rfc7348.txt>. Cited on page 83.

MCKEOWN, N. *Protocol-independent switch architecture*. 2015. Accessed: 2018-09-24. Available from Internet: <http://schd.ws/hosted_files/p4workshop2015/c9/NickM-P4-Workshop-June-04-2015.pdf>. Cited 5 times on pages 9, 23, 33, 37, and 38.

MCKEOWN, N.; ANDERSON, T.; BALAKRISHNAN, H.; PARULKAR, G.; PETERSON, L.; REXFORD, J.; SHENKER, S.; TURNER, J. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/1355734.1355746>. Cited 2 times on pages 22 and 32.

MEJIA, J. S.; FEFERMAN, D. L.; ROTHENBERG, C. E. Network address translation using a programmable dataplane processor. *Workshop em Desempenho de Sistemas Computacionais e de Comunicação (WPerformance_CSBC)*, v. 17, n. 1/2018, 2018. ISSN 2595-6167. Available from Internet: <http://portaldeconteudo.sbc.org.br/index.php/wperformance/article/view/3333>. Cited on page 27.

MONSANTO, C.; REICH, J.; FOSTER, N.; REXFORD, J.; WALKER, D. Composing software defined networks. In: *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. Lombard, IL: USENIX Association, 2013. (nsdi'13), p. 1–13. ISBN 978-1-931971-00-3. Available from Internet: <https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/monsanto>. Cited on page 38.

NETRONOME. [S.l.], 2015. Accessed: 2018-09-24. Available from Internet: <https://www.netronome.com/technology/p4/>. Cited on page 48.

OPENDATAPLANE. 2018. Accessed: 2018-09-24. Available from Internet: <http://www.opendataplane.org>. Cited 4 times on pages 23, 33, 35, and 147.

OPENDAYLIGHT. 2018. Accessed: 2018-09-24. Available from Internet: <https://www.opendaylight.org/>. Cited on page 72.

OPENSWITCH (OPX) Network Operating System. 2017. Accessed: 2018-09-24. Available from Internet: <http://www.openswitch.net>. Cited on page 48.

P. Bosshart et al. P4: Programming protocol-independent packet processors. *SIGCOMM Comput. Commun. Rev.*, ACM, New York, NY, USA, jul. 2014. ISSN 0146-4833. Cited 3 times on pages 23, 32, and 39.

P4 High Level Intermediate Representation. 2018. Accessed: 2018-09-24. Available from Internet: <https://github.com/p4lang/p4-hlir>. Cited 2 times on pages 52 and 56.

$P4_{14}$ SPEC. 2017. Accessed: 2018-09-24. Available from Internet: <https://p4lang.github.io/p4-spec/p4-14/v1.0.4/tex/p4.pdf>. Cited on page 44.

$P4_{16}$ SPEC. 2017. Accessed: 2018-09-24. Available from Internet: <https://p4lang.github.io/p4-spec/docs/P4-16-v1.0.0-spec.pdf>. Cited on page 44.

P4API, W. G. *P4 Runtime*. 2018. Accessed: 2018-09-24. Available from Internet: <https://s3-us-west-2.amazonaws.com/p4runtime/docs/master/P4Runtime-Spec.pdf>. Cited 2 times on pages 23 and 33.

PATRA, P. G.; ROTHENBERG, C. E.; PONGRáCZ, G. Macsad: Multi-architecture compiler system for abstract dataplanes (aka partnering p4 with odp). In: *Proceedings of the 2016 ACM SIGCOMM Conference*. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 623–624. ISBN 978-1-4503-4193-6. Available from Internet: <http://doi.acm.org/10.1145/2934872.2959077>. Cited 3 times on pages 25, 31, and 51.

PATRA, P. G.; ROTHENBERG, C. E.; PONGRACZ, G. Macsad: High performance dataplane applications on the move. In: *2017 IEEE 18th International Conference on High Performance Switching and Routing (HPSR)*. [S.l.: s.n.], 2017. p. 1–6. ISSN 2325-5609. Cited 3 times on pages 25, 31, and 51.

PATRA, P. G. K.; CESEN, F. E. R.; MEJIA, J. S.; FEFERMAN, D. L.; CSIKOR, L.; ROTHENBERG, C. E.; PONGRACZ, G. Towards a sweet spot of dataplane programmability, portability and performance: On the scalability of multi-architecture p4 pipelines. *IEEE Journal on Selected Areas in Communications*, p. 1–1, 2018. ISSN 0733-8716. Cited 5 times on pages 25, 27, 30, 31, and 51.

PFAFF, B.; PETTIT, J.; KOPONEN, T.; JACKSON, E. J.; ZHOU, A.; RAJAHALME, J.; GROSS, J.; WANG, A.; STRINGER, J.; SHELAR, P.; AMIDON, K.; CASADO, M. The design and implementation of open vswitch. In: *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2015. (NSDI'15), p. 117–130. ISBN 978-1-931971-218. Available from Internet: <http://dl.acm.org/citation.cfm?id=2789770.2789779>. Cited 2 times on pages 27 and 48.

RIZZO, L. netmap: A novel framework for fast packet i/o. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012. p. 101–112. ISBN 978-931971-93-5. Available from Internet: <https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo>. Cited on page 34.

RODRIGUEZ, F.; PATRA, P. G. K.; CSIKOR, L.; ROTHENBERG, C.; LAKI, P. V. S.; PONGRáCZ, G. Bb-gen: A packet crafter for p4 target evaluation. In: *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*. New York, NY, USA: ACM, 2018. (SIGCOMM '18), p. 111–113. ISBN 978-1-4503-5915-3. Available from Internet: <http://doi.acm.org/10.1145/3234200.3234229>. Cited 4 times on pages 26, 29, 30, and 77.

SAPIO, A.; BALDI, M.; PONGRáCZ, G. Cross-platform estimation of network function performance. In: *2015 Fourth European Workshop on Software Defined Networks*. [S.l.: s.n.], 2015. p. 73–78. ISSN 2379-0350. Cited on page 28.

SOFTWARE for Open Networking in the Cloud (SONiC). 2016. Accessed: 2018-09-24. Available from Internet: <https://azure.github.io/SONiC/>. Cited on page 48.

SONG, H. Protocol-oblivious forwarding: Unleash the power of sdn through a future-proof forwarding plane. In: *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*. New York, NY, USA: ACM, 2013. (HotSDN '13), p. 127–132. ISBN 978-1-4503-2178-5. Available from Internet: <http://doi.acm.org/10.1145/2491185.2491190>. Cited 3 times on pages 23, 32, and 38.

SUN, W.; RICCI, R. Fast and flexible: Parallel packet processing with gpus and click. In: *Proceedings of the Ninth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*. Piscataway, NJ, USA: IEEE Press, 2013. (ANCS '13), p. 25–36. ISBN 978-1-4799-1640-5. Available from Internet: <http://dl.acm.org/citation.cfm?id=2537857.2537861>. Cited on page 48.

T4P4S Git. 2016. Accessed: 2018-09-24. Available from Internet: <https://github.com/p4elte/t4p4s>. Cited on page 48.

VANVOORHIS, C. W.; MORGAN, B. L. Understanding power and rules of thumb for determining sample sizes. *Tutorials in Quantitative Methods for Psychology*, v. 3, n. 2, p. 43–50, 2007. Cited on page 120.

WANG, Y.; GOBRIEL, S.; WANG, R.; TAI, T.-Y. C.; DUMITRESCU, C. Hash table design and optimization for software virtual switches. In: *Proceedings of the 2018 Afternoon Workshop on Kernel Bypassing Networks*. New York, NY, USA: ACM, 2018. (KBNets'18), p. 22–28. ISBN 978-1-4503-5909-2. Available from Internet: <http://doi.acm.org/10.1145/3229538.3229542>. Cited on page 28.

ANNEX A – Layer 2 Forwarding (L2FWD)

A.1 L2FWD $P4_{14}$ Program

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

header ethernet_t ethernet;

parser start {
    return parse_ethernet;
}

parser parse_ethernet {
    extract(ethernet);
    return ingress;
}

action _drop() {
    drop();
}

action _nop() {
}

#define MAC_LEARN_RECEIVER 1024

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

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

table smac {
    reads {
        ethernet.srcAddr : exact;
```

```
    }
    actions {mac_learn; _nop;}
    size : 512;
}

action forward(port) {
    modify_field(standard_metadata.egress_port, port);
}

action bcast() {
    modify_field(standard_metadata.egress_port, 100);
}

table dmac {
    reads {
        ethernet.dstAddr : exact;
    }
    actions {forward; bcast;}
    size : 512;
}

control ingress {
    apply(smac);
    apply(dmac);
}

control egress {
}
```

Listing A.1 – L2FWD $P4_{14}$ code

A.2 L2FWD $P4_{16}$ Program

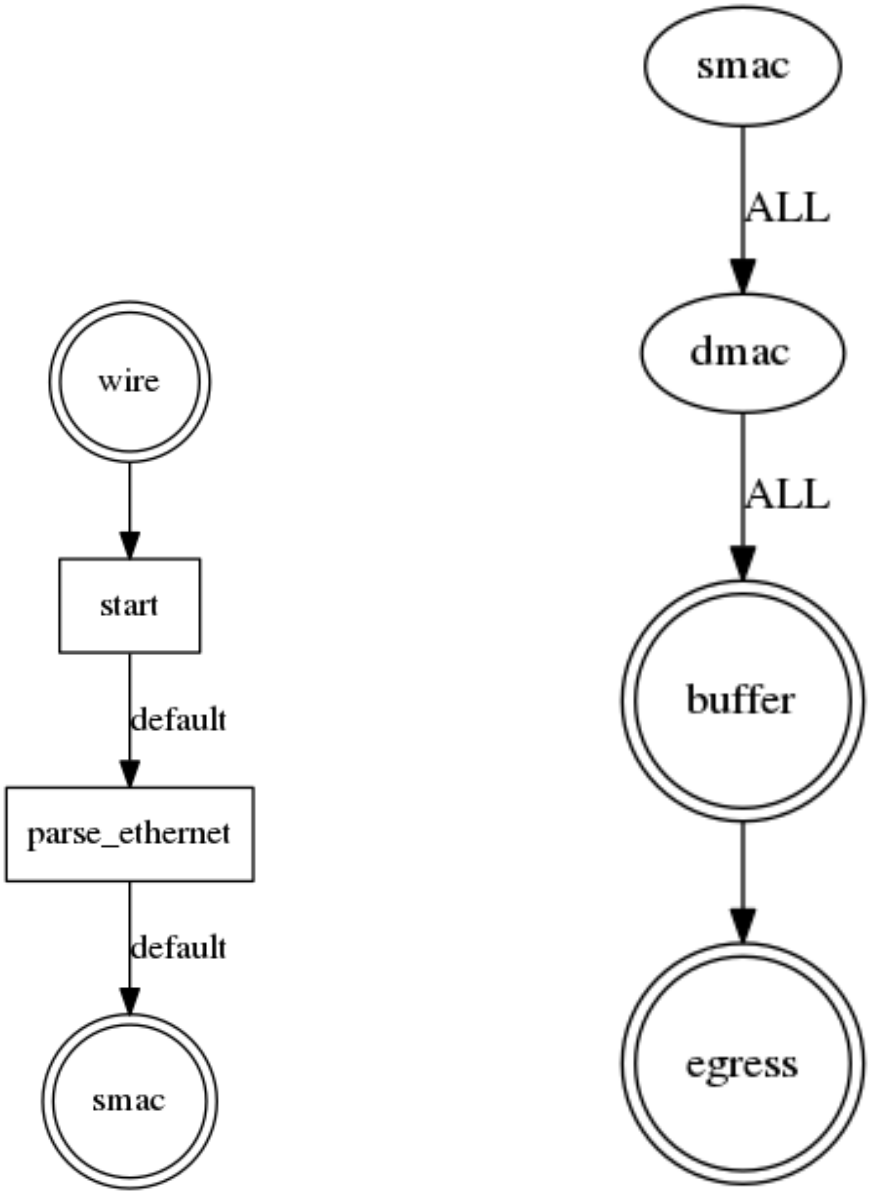
```
#include <core.p4>
#include <v1model.p4>

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

struct metadata { }

struct headers {
    @name(".ethernet")
    ethernet_t ethernet;
}
```

```
16
17  parser ParserImpl(packet_in packet, out headers hdr, inout metadata meta,
        inout standard_metadata_t standard_metadata) {
18      @name(".parse_ethernet") state parse_ethernet {
19          packet.extract(hdr.ethernet);
20          transition accept;
21      }
22
23      @name(".start") state start {
24          transition parse_ethernet;
25      }
26  }
27
28  control egress(inout headers hdr, inout metadata meta, inout
        standard_metadata_t standard_metadata) {
29      apply {
30      }
31  }
32
33  @name("mac_learn_digest") struct mac_learn_digest {
34      bit<48> srcAddr;
35      bit<9>  ingress_port;
36  }
37
38  control ingress(inout headers hdr, inout metadata meta, inout
        standard_metadata_t standard_metadata) {
39      @name(".forward") action forward(bit<9> port) {
40          standard_metadata.egress_port = port;
41      }
42
43      @name(".bcast") action bcast() {
44          standard_metadata.egress_port = 9w100;
45      }
46
47      @name(".mac_learn") action mac_learn() {
48          digest<mac_learn_digest>((bit<32>)1024, { hdr.ethernet.srcAddr,
        standard_metadata.ingress_port });
49      }
50
51      @name("._nop") action _nop() {
52      }
53
54      @name(".dmac") table dmac {
55          actions = {
56              forward;
57              bcast;
58          }
```

```
59
60          key = {
61              hdr.ethernet.dstAddr: exact;
62          }
63          size = 512;
64      }
65
66      @name(".smac") table smac {
67          actions = {
68              mac_learn;
69              _nop;
70          }
71
72          key = {
73              hdr.ethernet.srcAddr: exact;
74          }
75          size = 512;
76      }
77
78      apply {
79          smac.apply();
80          dmac.apply();
81      }
82  }
83
84  control DeparserImpl(packet_out packet, in headers hdr) {
85      apply {
86          packet.emit(hdr.ethernet);
87      }
88  }
89
90  control verifyChecksum(inout headers hdr, inout metadata meta) {
91      apply {
92      }
93  }
94
95  control computeChecksum(inout headers hdr, inout metadata meta) {
96      apply {
97      }
98  }
99
100 V1Switch(ParserImpl(), verifyChecksum(), ingress(), egress(),
    computeChecksum(), DeparserImpl()) main;
```

Listing A.2 – L2FWD $P4_{16}$ code

A.3 Dependency Graphs for L2FWD Use Case



(a) L2FWD Parser Flow (b) L2FWD Table Flow

Figure 47 – L2FWD Dependency Graphs

ANNEX B – Layer 3 Forwarding (L3FWD)

B.1 L3FWDv4 $P4_{14}$ Program

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

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

parser start {
    return parse_ethernet;
}

#define ETHERTYPE_IPV4 0x0800

header ethernet_t ethernet;

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

header ipv4_t ipv4;
```

```
41
42  parser parse_ipv4 {
43      extract(ipv4);
44      return ingress;
45  }
46
47  action on_miss() {
48      drop () ;
49  }
50
51  action fib_hit_nexthop(dmac, port) {
52      modify_field(ethernet.dstAddr, dmac);
53      modify_field(standard_metadata.egress_port, port);
54      add_to_field(ipv4.ttl, -1);
55  }
56
57  table ipv4_fib_lpm {
58      reads {
59          ipv4.dstAddr : lpm;
60      }
61
62      actions {
63          fib_hit_nexthop;
64          on_miss;
65      }
66      size : 512;
67  }
68
69  action rewrite_src_mac(smac) {
70      modify_field(ethernet.srcAddr, smac);
71  }
72
73  table sendout {
74      reads {
75          standard_metadata.egress_port : exact;
76      }
77
78      actions {
79          on_miss;
80          rewrite_src_mac;
81      }
82      size : 512;
83  }
84
85  control ingress {
86      apply(ipv4_fib_lpm);
87      apply(sendout);
```

```
88 }
89
90 control egress {
91 }
```

Listing B.1 – L3FWDv4 $P4_{14}$ code

B.2 L3FWDv6 $P4_{14}$ Program

```
1  header_type ethernet_t {
2      fields {
3      dstAddr : 48;
4      srcAddr : 48;
5      etherType : 16;
6      }
7  }
8
9  header_type ipv6_t {
10   fields {
11     version : 4;
12     trafficClass : 8;
13     flowLabel : 20;
14     payloadLen : 16;
15     nextHdr : 8;
16     hopLimit : 8;
17     srcAddr : 128;
18     dstAddr : 128;
19   }
20 }
21
22 parser start {
23     return parse_ethernet;
24 }
25
26 header ethernet_t ethernet;
27
28 parser parse_ethernet {
29     extract(ethernet);
30     return select(latest.etherType) {
31         0x86DD : parse_ipv6;
32         default: ingress;
33     }
34 }
35
36 header ipv6_t ipv6;
37
38 parser parse_ipv6 {
39     extract(ipv6);
```

```
40      return ingress;
41  }
42
43  action on_miss() {
44      drop () ;
45  }
46
47  action fib_hit_nexthop(dmac, port) {
48      modify_field(ethernet.dstAddr, dmac);
49      modify_field(standard_metadata.egress_port, port);
50      add_to_field(ipv6.hopLimit, -1);
51  }
52
53  table ipv6_fib_lpm {
54      reads {
55          ipv6.dstAddr : lpm;
56      }
57      actions {
58          fib_hit_nexthop;
59          on_miss;
60      }
61      size : 512;
62  }
63
64  action rewrite_src_mac(smac) {
65      modify_field(ethernet.srcAddr, smac);
66  }
67
68  table sendout {
69      reads {
70          standard_metadata.egress_port : exact;
71      }
72      actions {
73          on_miss;
74          rewrite_src_mac;
75      }
76      size : 512;
77  }
78
79  control ingress {
80      apply(ipv6_fib_lpm);
81      apply(sendout);
82  }
83
84  control egress {
85  }
```

Listing B.2 – L3FWDv6 $P4_{14}$ code

B.3 L3FWDv4 $P4_{16}$ Program

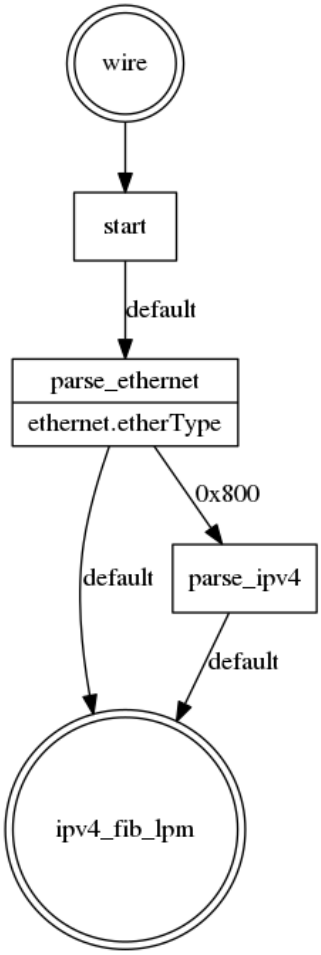
```
#include <core.p4>
#include <v1model.p4>

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

header ipv4_t {
    bit<8>  versionIhl;
    bit<8>  diffserv;
    bit<16> totalLen;
    bit<16> identification;
    bit<16> fragOffset;
    bit<8>  ttl;
    bit<8>  protocol;
    bit<16> hdrChecksum;
    bit<32> srcAddr;
    bit<32> dstAddr;
}

struct metadata {
}

struct headers {
    @name(".ethernet")
    ethernet_t ethernet;
    @name(".ipv4")
    ipv4_t ipv4;
}

parser ParserImpl(packet_in packet, out headers hdr, inout metadata meta,
    inout standard_metadata_t standard_metadata) {
    @name(".parse_ethernet") state parse_ethernet {
        packet.extract(hdr.ethernet);
        transition select(hdr.ethernet.etherType) {
            16w0x800: parse_ipv4;
            default: accept;
        }
    }

    @name(".parse_ipv4") state parse_ipv4 {
        packet.extract(hdr.ipv4);
        transition accept;
    }
```

```
46
47      @name(".start") state start {
48          transition parse_ethernet;
49      }
50  }
51
52  control egress(inout headers hdr, inout metadata meta, inout
        standard_metadata_t standard_metadata) {
53      apply {
54      }
55  }
56
57  control ingress(inout headers hdr, inout metadata meta, inout
        standard_metadata_t standard_metadata) {
58      @name(".fib_hit_nexthop") action fib_hit_nexthop(bit<48> dmac, bit<9>
        port) {
59          hdr.ethernet.dstAddr = dmac;
60          standard_metadata.egress_port = port;
61          hdr.ipv4.ttl = hdr.ipv4.ttl + 8w255;
62      }
63
64      @name(".on_miss") action on_miss() {
65          drop () ;
66      }
67
68      @name(".rewrite_src_mac") action rewrite_src_mac(bit<48> smac) {
69          hdr.ethernet.srcAddr = smac;
70      }
71
72      @name(".ipv4_fib_lpm") table ipv4_fib_lpm {
73          actions = {
74              fib_hit_nexthop;
75              on_miss;
76          }
77
78          key = {
79              hdr.ipv4.dstAddr: lpm;
80          }
81          size = 512;
82      }
83
84      @name(".sendout") table sendout {
85          actions = {
86              on_miss;
87              rewrite_src_mac;
88          }
89
```

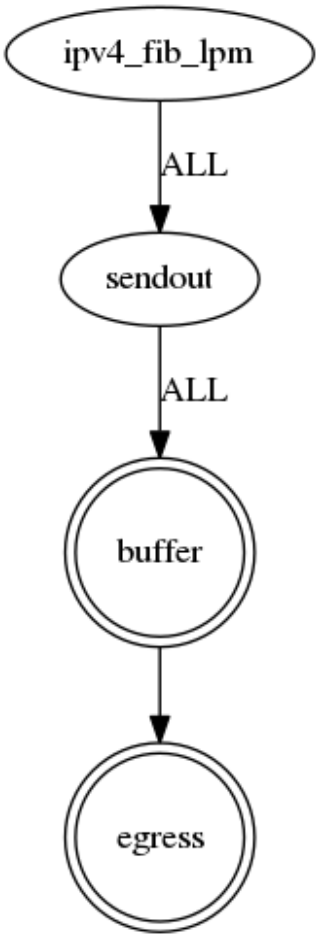
```
90          key = {
91              standard_metadata.egress_port: exact;
92          }
93          size = 512;
94      }
95
96      apply {
97          ipv4_fib_lpm.apply();
98          sendout.apply();
99      }
100 }
101
102 control DeparserImpl(packet_out packet, in headers hdr) {
103     apply {
104         packet.emit(hdr.ethernet);
105         packet.emit(hdr.ipv4);
106     }
107 }
108
109 control verifyChecksum(inout headers hdr, inout metadata meta) {
110     apply {
111     }
112 }
113
114 control computeChecksum(inout headers hdr, inout metadata meta) {
115     apply {
116     }
117 }
118
119 V1Switch(ParserImpl(), verifyChecksum(), ingress(), egress(),
    computeChecksum(), DeparserImpl()) main;
```

Listing B.3 – L3FWDv4 $P4_{16}$ Code

B.4 Dependency Graphs for L3FWDv4 Use Case



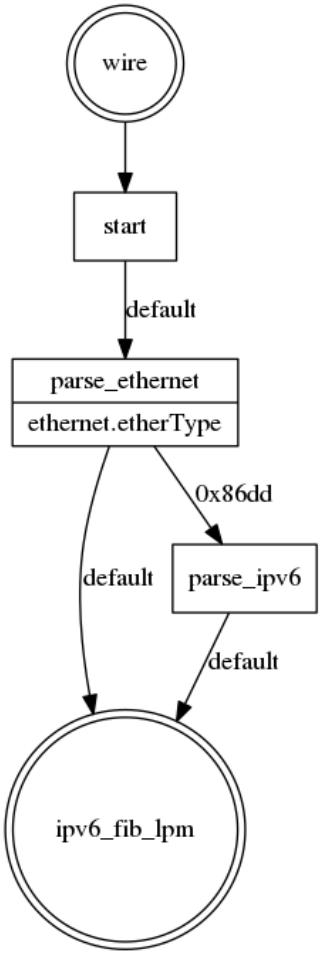
(a) L3FWDv4 Parser Flow



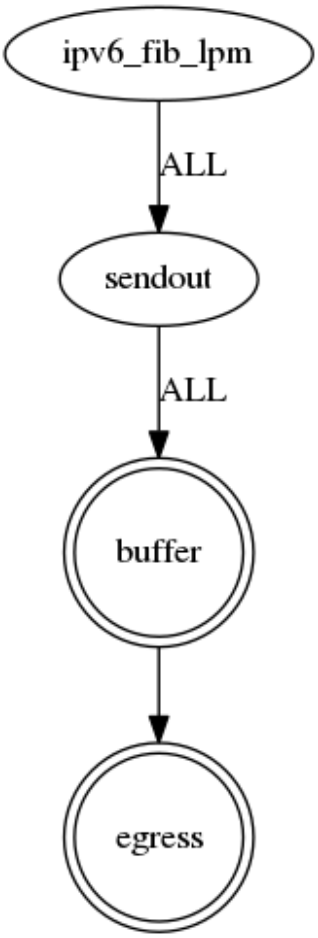
(b) L3FWDv4 Table Flow

Figure 48 – L3FWDv4 Dependency Graphs

B.5 Dependency Graphs for L3FWDv6 Use Case



(a) L3FWDv6 Parser Flow



(b) L3FWDv6 Table Flow

Figure 49 – L3FWDv6 Dependency Graphs

ANNEX C – Network Address Translation (NAT)

C.1 NAT $P4_{14}$ Program

```
#define ETHERTYPE_ARP  0x0806
#define ETHERTYPE_IPV4 0x0800
#define MAC_LEARN_RECEIVER 1024
#define IP_PROT_TCP 0x06
#define IP_PROT_UDP 0x11

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

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

header_type tcp_t {
    fields {
        srcPort : 16;
        dstPort : 16;
        seqNo : 32;
        ackNo : 32;
        dataOffset : 4;
```

```
39          res : 4;
40          flags : 8;
41          window : 16;
42          checksum : 16;
43          urgentPtr : 16;
44      }
45  }
46
47  header ethernet_t ethernet;
48  header ipv4_t ipv4;
49  header tcp_t tcp;
50
51  /************   Metadata   ************/
52  header_type routing_metadata_t {
53      fields {
54          is_int_if : 8;
55      }
56  }
57  metadata routing_metadata_t routing_metadata;
58
59  /************   Parser   ************/
60  parser start {
61      return parse_ethernet;
62  }
63
64  parser parse_ethernet {
65      extract(ethernet);
66      return select(latest.etherType) {
67          ETHERTYPE_IPV4 : parse_ipv4;
68          default: ingress;
69      }
70  }
71
72  parser parse_ipv4 {
73      extract(ipv4);
74      return select(ipv4.protocol) {
75          IP_PROT_TCP : parse_tcp;
76          default: ingress;
77      }
78  }
79
80  parser parse_tcp {
81      extract(tcp);
82      return ingress;
83  }
84
85  field_list mac_learn_digest {
```

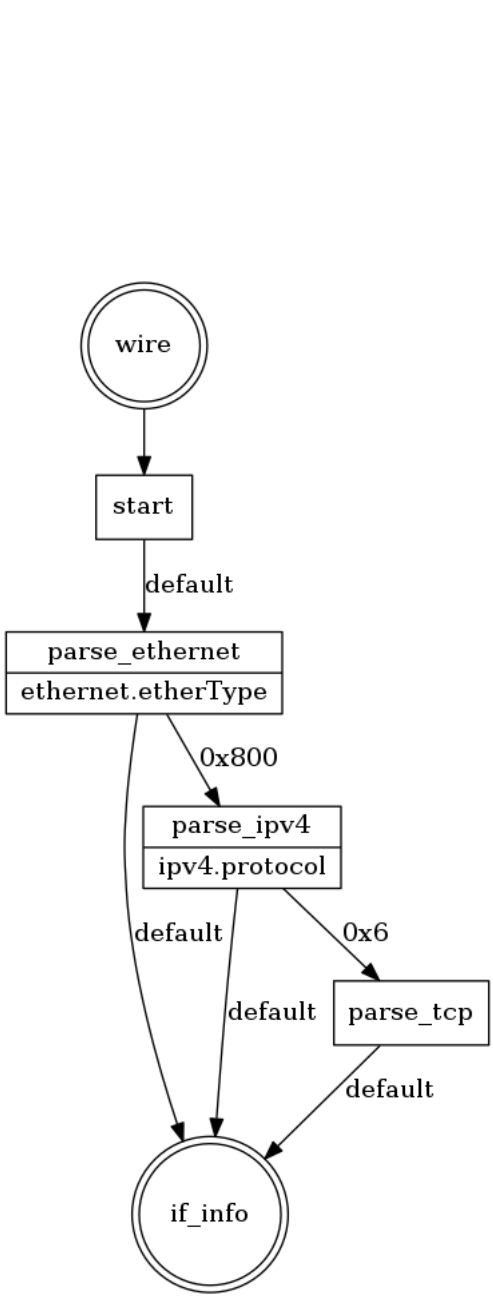
```
86      ethernet.srcAddr;
87      standard_metadata.ingress_port;
88  }
89
90  field_list natTcp_learn_digest {
91      ipv4.srcAddr;
92      tcp.srcPort;
93  }
94
95  /************   Ingress Processing   *************/
96  action _drop() {
97      drop();
98  }
99
100 action _nop() {
101 }
102
103 /*************   set IF info and others   *************/
104 action set_if_info(is_int) {
105     modify_field(routing_metadata.is_int_if, is_int);
106 }
107
108 table if_info {
109     reads {
110         standard_metadata.ingress_port : exact;
111     }
112     actions {set_if_info; _drop;}
113     size : 512;
114 }
115
116 /*************   process mac learn   *************/
117 action mac_learn() {
118     generate_digest(MAC_LEARN_RECEIVER, mac_learn_digest);
119 }
120
121 table smac {
122     reads {
123         ethernet.srcAddr : exact;
124     }
125     actions {mac_learn; _nop;}
126     size : 512;
127 }
128
129 /*************   Nat control   *************/
130 action natTcp_learn() {
131     generate_digest(MAC_LEARN_RECEIVER, natTcp_learn_digest);
132 }
```

```
133
134 action nat_hit_int_to_ext(srcAddr) {
135     modify_field(ipv4.srcAddr, srcAddr);
136 }
137
138 table nat_up {
139     reads {
140         ipv4.srcAddr: lpm;
141     }
142     actions {
143         nat_hit_int_to_ext;
144         natTcp_learn;
145     }
146     size: 512;
147 }
148
149 action nat_hit_ext_to_int(dstAddr) {
150     modify_field(ipv4.dstAddr, dstAddr);
151 }
152
153 table nat_dw {
154     reads {
155         tcp.dstPort : exact;
156     }
157     actions {
158         nat_hit_ext_to_int;
159         _drop;
160     }
161     size: 512;
162 }
163
164 /*************   Forwarding ipv4  *************/
165 action set_nhop(port, dstAddr) {
166     modify_field(standard_metadata.egress_port, port);
167     modify_field(ethernet.dstAddr, dstAddr);
168 }
169
170 table ipv4_lpm {
171     reads {
172         ipv4.dstAddr : lpm;
173     }
174     actions {
175         set_nhop;
176         _drop;
177     }
178     size: 512;
179 }
```

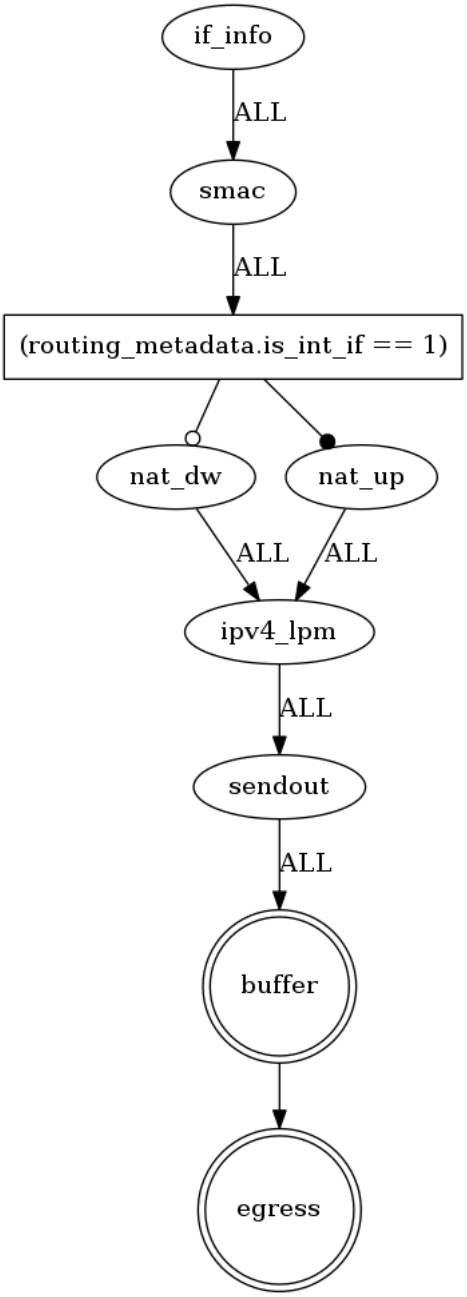
```
180
181 action rewrite_src_mac(srcAddr) {
182     modify_field(ethernet.srcAddr, srcAddr);
183 }
184
185 table sendout {
186     reads {
187         standard_metadata.egress_port: exact;
188     }
189     actions {
190         rewrite_src_mac;
191         _drop;
192     }
193     size: 512;
194 }
195
196 /*************    Apply    *************/
197 control ingress {
198     apply(if_info);
199     apply(smac);
200     if (routing_metadata.is_int_if == 1) {
201         apply(nat_up);
202     } else {
203         apply(nat_dw);
204     }
205     apply(ipv4_lpm);
206     apply(sendout);
207 }
208
209 control egress {
210 }
```

Listing C.1 – NAT $P4_{14}$ code

C.2 Dependency Graphs for NAT Use Case



(a) NAT Parser Flow



(b) NAT Table Flow

Figure 50 – NAT Dependency Graphs

ANNEX D – Data Center Gateway (DCG)

D.1 DCG $P4_{14}$ Program

```
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;

```

```
41 header_type vxlan_t {
42     fields {
43         flags : 8;
44         reserved : 24;
45         vni : 24;
46         reserved2 : 8;
47     }
48 }
49
50 header vxlan_t vxlan;
51
52 header_type arp_t {
53     fields {
54         htype : 16;
55         ptype : 16;
56         hlength : 8;
57         plength: 8;
58         opcode: 16;
59     }
60 }
61
62 header arp_t arp;
63 header ethernet_t inner_ethernet;
64 header ipv4_t inner_ipv4;
65
66 /************   Parser   *************/
67 #define MAC_LEARN_RECEIVER      1024
68 #define ETHERTYPE_IPV4          0x0800
69 #define ETHERTYPE_ARP           0x0806
70
71 #define IP_PROTOCOLS_IPHL_UDP   0x511
72 #define UDP_PORT_VXLAN          4789
73
74 #define BONE                    1
75 #define BTWO                    2
76 #define BTHREE                  3
77 #define BIT_WIDTH               16
78
79 parser start {
80     return parse_ethernet;
81 }
82
83 parser parse_ethernet {
84     extract(ethernet);
85     return select(latest.etherType) {
86         ETHERTYPE_IPV4 : parse_ipv4;
87         ETHERTYPE_ARP : parse_arp;
```

```
88          default: ingress;
89      }
90  }
91
92  parser parse_arp{
93      extract(arp);
94      return ingress;
95  }
96
97  parser parse_ipv4 {
98      extract(ipv4);
99      return select(latest.fragOffset, latest.ihl, latest.protocol) {
100         IP_PROTOCOLS_IPHL_UDP : parse_udp;
101         default: ingress;
102     }
103 }
104
105 parser parse_udp {
106     extract(udp);
107     return select (latest.dstPort) {
108         UDP_PORT_VXLAN : parse_vxlan;
109         default : ingress;
110     }
111 }
112
113 parser parse_vxlan {
114     extract(vxlan);
115     return parse_inner_ethernet;
116 }
117
118 parser parse_inner_ethernet {
119     extract(inner_ethernet);
120     return select(latest.etherType) {
121         ETHERTYPE_IPV4 : parse_inner_ipv4;
122         default: ingress;
123     }
124 }
125
126 parser parse_inner_ipv4 {
127     extract(inner_ipv4);
128     return ingress;
129 }
130
131 /************   Actions   ************/
132 action _drop() {
133     drop();
134 }
```

```
135
136 action _nop() {
137 }
138
139 field_list ipv4_checksum_list {
140     ipv4.version;
141     ipv4.ihl;
142     ipv4.diffserv;
143     ipv4.totalLen;
144     ipv4.identification;
145     ipv4.flags;
146     ipv4.fragOffset;
147     ipv4.ttl;
148     ipv4.protocol;
149     ipv4.srcAddr;
150     ipv4.dstAddr;
151 }
152
153 field_list mac_learn_digest {
154     ethernet.srcAddr;
155     routing_metadata.ingress_port;
156 }
157
158 field_list inner_ipv4_checksum_list {
159     inner_ipv4.version;
160     inner_ipv4.ihl;
161     inner_ipv4.diffserv;
162     inner_ipv4.totalLen;
163     inner_ipv4.identification;
164     inner_ipv4.flags;
165     inner_ipv4.fragOffset;
166     inner_ipv4.ttl;
167     inner_ipv4.protocol;
168     inner_ipv4.srcAddr;
169     inner_ipv4.dstAddr;
170 }
171
172 field_list_calculation inner_ipv4_checksum {
173     input {
174         inner_ipv4_checksum_list;
175     }
176
177     algorithm : csum16;
178     output_width : 16;
179 }
180
181 action mac_learn() {
```

```
182     generate_digest(MAC_LEARN_RECEIVER, mac_learn_digest);
183 }
184
185 table MAClearn {
186     reads {
187         ethernet.srcAddr : exact;
188     }
189
190     actions {
191         mac_learn;
192         _nop;
193     }
194
195     size : 512;
196 }
197
198 header_type routing_metadata_t {
199     fields {
200         outport: 2;
201         res: 2;
202         aux : 2;
203         egress_port : 2;
204         ingress_port : 8;
205         lb_hash: 16;
206     }
207 }
208
209 metadata routing_metadata_t routing_metadata;
210
211 action forward(port, nhop, mac) {
212     modify_field(standard_metadata.egress_port, port);
213     modify_field(ethernet.dstAddr, mac);
214     modify_field(routing_metadata.res, BTHREE);
215 }
216
217 action Tcast() {
218     modify_field(routing_metadata.res, BONE);
219 }
220
221 action Tmac() {
222     modify_field(routing_metadata.res, BTWO);
223 }
224
225 table MACfwd {
226     reads {
227         ethernet.dstAddr : exact;
228     }
```

```
229
230     actions {
231         forward;
232         _drop;
233         Tcast;
234         Tmac;
235     }
236
237     size : 512;
238 }
239
240 table ownMAC{
241     reads {
242         ethernet.srcAddr : exact;
243     }
244
245     actions {
246         _nop;
247         forward;
248     }
249 }
250
251 action arp() {
252     generate_digest(ETHERTYPE_ARP, mac_learn_digest);
253     modify_field(routing_metadata.res, BTWO);
254 }
255
256 table ARPselect {
257     reads {
258         ethernet.etherType: exact;
259     }
260
261     actions {
262         arp;
263         _nop;
264     }
265
266     size : 2;
267 }
268
269 action balancer(){
270     modify_field(routing_metadata.aux, BONE);
271     modify_field(routing_metadata.lb_hash,1);
272 }
273
274 action _pop(){
275     modify_field(routing_metadata.aux, BTWO);
```

```
276 }
277
278 action jump(){
279     modify_field(routing_metadata.aux, BTHREE);
280 }
281
282 table LBselector{
283     reads {
284         ipv4.dstAddr : exact;
285     }
286
287     actions {
288         jump;
289         _pop;
290         balancer;
291     }
292
293     size: 128;
294 }
295
296 action _pop_vxlan(){
297     remove_header(ethernet);
298     remove_header(ipv4);
299     remove_header(vxlan);
300     modify_field(udp.dstPort, 700);
301 }
302
303 table vpop{
304     reads {
305         ipv4.srcAddr : exact;
306     }
307
308     actions {
309         _pop_vxlan;
310         _nop;
311     }
312 }
313
314 action press(vnid, nhop, srcAddr){
315     add_header(vxlan);
316     add_header(udp);
317     add_header(inner_ipv4);
318     copy_header(inner_ipv4, ipv4);
319     add_header(inner_ethernet);
320     copy_header(inner_ethernet, ethernet);
321
322     modify_field(ipv4.dstAddr, nhop);
```

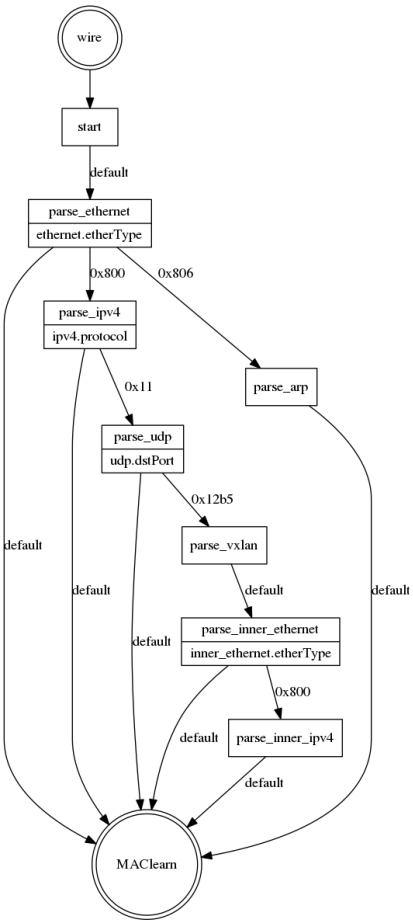
```
323     modify_field(ipv4.srcAddr, srcAddr);
324     modify_field(ipv4.protocol, 0x11);
325     modify_field(ipv4.ttl, 64);
326     modify_field(ipv4.version, 0x4);
327     modify_field(ipv4.ihl, 0x5);
328     modify_field(ipv4.identification, 0);
329     modify_field(inner_ipv4.totalLen, ipv4.totalLen);
330     modify_field(ethernet.etherType, ETHERTYPE_IPV4);
331     modify_field(udp.dstPort, UDP_PORT_VXLAN);
332     modify_field(udp.checksum, 0);
333     modify_field(udp.length_, ipv4.totalLen + 30);
334     modify_field(vxlan.flags, 0x8);
335     modify_field(vxlan.reserved, 0);
336     modify_field(vxlan.vni, vnid);
337     modify_field(vxlan.reserved2, 0);
338 }
339
340 table LB{
341     reads {
342         ipv4.srcAddr : exact;
343     }
344
345     actions {
346         press;
347         _nop;
348     }
349
350     size:1024;
351 }
352
353 action nhop_ipv4(nhop_ipv4) {
354     modify_field(ipv4.dstAddr, nhop_ipv4);
355 }
356
357 table LBipv4 {
358     reads {
359         routing_metadata.lb_hash : exact;
360     }
361
362     actions {
363         nhop_ipv4;
364         _nop;
365     }
366
367     size:1024;
368 }
369
```

```
370 action nhop(port, dmac){
371     modify_field(standard_metadata.egress_port, port);
372     modify_field(ethernet.dstAddr, dmac);
373     modify_field(ipv4.ttl,ipv4.ttl - 1);
374 }
375
376 table L3{
377     reads {
378         inner_ipv4.dstAddr : lpm;
379     }
380
381     actions {
382         nhop;
383         _nop;
384     }
385 }
386
387 action rewrite_src_mac(smac) {
388     modify_field(ethernet.srcAddr, smac);
389 }
390
391 table sendout {
392     reads {
393         standard_metadata.egress_port : exact;
394     }
395
396     actions {
397         _nop;
398         rewrite_src_mac;
399     }
400
401     size : 512;
402 }
403
404 /************   Control   ************/
405 control ingress {
406     apply(MAClearn);
407     apply(MACfwd);
408     if (routing_metadata.res == BONE){
409         apply(ARPselect);
410     }
411     else if (routing_metadata.res == BIWO){
412         apply(ownMAC);
413         apply(LBselector);
414
415         if (routing_metadata.aux == BONE){
416             apply(LB);
```

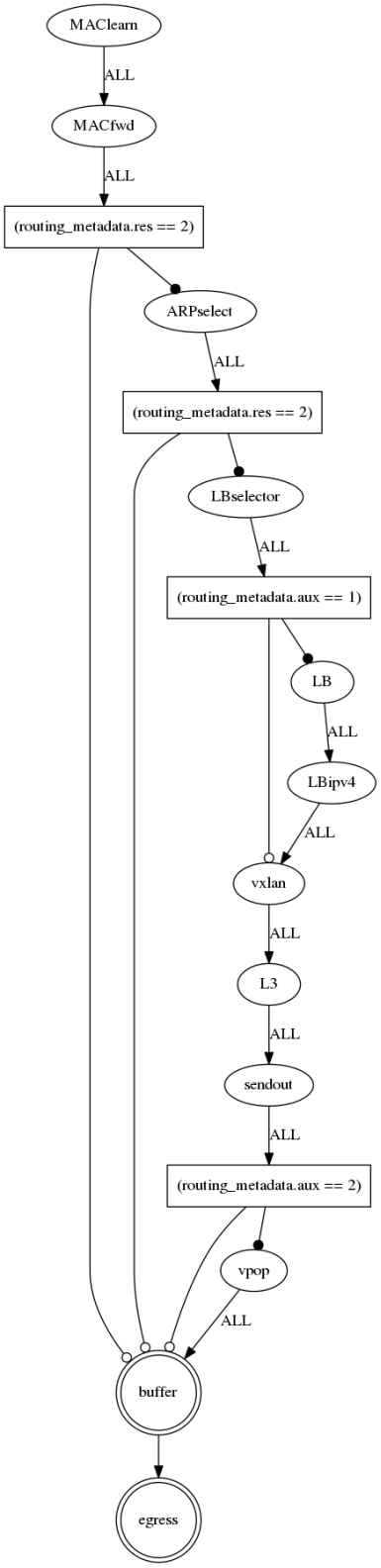
```
417             apply(LBipv4);
418         }
419
420         apply(L3);
421         apply(sendout);
422         if (routing_metadata.aux == BTWO){
423             apply(vpop);
424         }
425     }
426 }
427
428 control egress {
429 }
```

Listing D.1 – DCG $P4_{14}$ Code

D.2 Dependency Graphs for DCG Use Case



(a) DCG Parser Flow



(b) DCG Table Flow

Figure 51 – DCG Dependency Graphs

ANNEX E – Broadband Network Gateway (BNG)

E.1 BNG $P4_{16}$ Program

```
header cpu_header_t {
    bit<64> preamble;
    bit<8>  device;
    bit<8>  reason;
    bit<8>  if_index;
}

header arp_t {
    bit<16> htype;
    bit<16> ptype;
    bit<8>  hlen;
    bit<8>  plen;
    bit<16> oper;
}

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

header ipv4_t {
    bit<4>      version;
    bit<4>      ihl;
    bit<8>      diffserv;
    bit<16>     totalLen;
    bit<16>     identification;
    bit<3>      flags;
    bit<13>     fragOffset;
    bit<8>      ttl;
    bit<8>      protocol;
    bit<16>     hdrChecksum;
    bit<32>     srcAddr;
    bit<32>     dstAddr;
}

header icmp_t {
    bit<8>  type;
```

```
39      bit<8>  code;
40      bit<16> checksum;
41  }
42
43  header tcp_t {
44      bit<16> srcPort;
45      bit<16> dstPort;
46      bit<32> seqNo;
47      bit<32> ackNo;
48      bit<4> dataOffset;
49      bit<4> res;
50      bit<8> flags;
51      bit<16> window;
52      bit<16> checksum;
53      bit<16> urgentPtr;
54  }
55
56  header gre_t {
57      bit<1> C;
58      bit<1> R;
59      bit<1> K;
60      bit<1> S;
61      bit<1> s;
62      bit<3> recurse;
63      bit<5> flags;
64      bit<3> ver;
65      bit<16> proto;
66  }
67
68  header udp_h {
69      bit<16> srcPort;
70      bit<16> dstPort;
71      bit<16> length_;
72      bit<16> checksum;
73  }
74
75  header sctp_h {
76      bit<16> srcPort;
77      bit<16> dstPort;
78      bit<32> verifTag;
79      bit<32> checksum;
80  }
81
82  header arp_ipv4_t {
83      bit<48>  sha;
84      bit<32>  spa;
85      bit<48>  tha;
```

```
    bit<32>  tpa;
}
```

Listing E.1 – BNG Header Details

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

/************   Constants   ************/
const bit<16> ETHERTYPE_IPV4 = 0x0800;
const bit<16> ETHERTYPE_ARP  = 0x0806;
const bit<8>  IPPROTO_ICMP   = 0x01;

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

struct headers {
    ethernet_t   ethernet;
    ethernet_t   outer_ethernet;
    ethernet_t   ethernet_decap;
    arp_t        arp;
    ipv4_t       ipv4;
    ipv4_t       outer_ipv4;
    gre_t        gre;
    tcp_t        tcp;
    icmp_t       icmp;
    @name("inner_ipv4")
    ipv4_t       inner_ipv4;
    @name("inner_ethernet")
    ethernet_t   inner_ethernet;
    @name("inner_tcp")
    tcp_t        inner_tcp;
    @name("inner_icmp")
    icmp_t       inner_icmp;
}

struct meta_ipv4_t {
     bit<4>       version;
     bit<4>       ihl;
     bit<8>       diffserv;
     bit<16>      totalLen;
     bit<16>      identification;
     bit<3>       flags;
     bit<13>      fragOffset;
```

```
44      bit<8>          ttl;
45      bit<8>          protocol;
46      bit<16>         hdrChecksum;
47      bit<32>         srcAddr;
48      bit<32>         dstAddr;
49  }
50
51  /************   Metadata   ************/
52  struct routing_metadata_t {
53      bit<32> nhgroup;
54      bit<32> dst_ipv4;
55      bit<32> src_ipv4;
56      bit<48>  mac_da;
57      bit<48>  mac_sa;
58      bit<9>   egress_port;
59      bit<48>  my_mac;
60
61      bit<32> nhop_ipv4;
62      bit<1>  do_forward;
63      bit<1>  rewrite_outer;
64      bit<16> tcp_sp;
65      bit<16> tcp_dp;
66
67      bit<8>  if_index;
68      bit<32> if_ipv4_addr;
69      bit<48> if_mac_addr;
70      bit<1>  is_ext_if;
71
72      bit<32> tunnel_id;
73      bit<5>  ingress_tunnel_type;
74      bit<1>  tcp_inner_en;
75      bit<16> lkp_inner_l4_sport;
76      bit<16> lkp_inner_l4_dport;
77
78      bit<32>  dst_inner_ipv4;
79      bit<32>  src_inner_ipv4;
80
81      bit<32> meter_tag;
82  }
83
84  struct metadata {
85      @name(".routing_metadata")
86      routing_metadata_t routing_metadata;
87      @name(".meta_ipv4")
88      meta_ipv4_t meta_ipv4 ;
89  }
90
```

```
91  /************   Parser   *************/
92  parser ParserImpl(packet_in packet, out headers hdr, inout metadata meta,
        inout standard_metadata_t standard_metadata) {
93      @name(".start") state start {
94          transition parse_ethernet;
95      }
96
97      @name ("parse_ethernet") state parse_ethernet {
98          packet.extract(hdr.ethernet);
99          transition select(hdr.ethernet.etherType) {
100             ETHERTYPE_IPV4 : parse_ipv4;
101             ETHERTYPE_ARP  : parse_arp;
102             default        : accept;
103         }
104     }
105
106     @name ("parse_arp") state parse_arp {
107         packet.extract(hdr.arp);
108         transition accept;
109     }
110
111     @name("parse_ipv4") state parse_ipv4 {
112         packet.extract(hdr.ipv4);
113         transition select(hdr.ipv4.protocol) {
114             IPPROTO_ICMP : parse_icmp;
115             8w0x6 :         parse_tcp;
116             8w47 :         parse_gre;
117             default      : accept;
118         }
119     }
120
121     @name("parse_icmp") state parse_icmp {
122         packet.extract(hdr.icmp);
123         transition accept;
124     }
125
126     @name("parse_tcp") state parse_tcp {
127         packet.extract<tcp_t>(hdr.tcp);
128         transition accept;
129     }
130
131     @name("parse_gre") state parse_gre {
132         packet.extract<gre_t>(hdr.gre);
133         transition select(hdr.gre.C, hdr.gre.R, hdr.gre.K, hdr.gre.S, hdr.
        gre.s, hdr.gre.recurse, hdr.gre.flags, hdr.gre.ver, hdr.gre.proto) {
134             (1w0x0, 1w0x0, 1w0x0, 1w0x0, 1w0x0, 3w0x0, 5w0x0, 3w0x0, 16
        w0x800): parse_gre_ipv4;
```

```
135             default: accept;
136         }
137     }
138
139     @name(".parse_gre_ipv4") state parse_gre_ipv4 {
140         transition parse_inner_ipv4;
141     }
142
143     @name(".parse_inner_ipv4") state parse_inner_ipv4 {
144         packet.extract(hdr.inner_ipv4);
145         transition select(hdr.inner_ipv4.fragOffset, hdr.inner_ipv4.ihl,
    hdr.inner_ipv4.protocol) {
146             (13w0x0, 4w0x5, 8w0x1): parse_inner_icmp;
147             (13w0x0, 4w0x5, 8w0x6): parse_inner_tcp;
148             default: accept;
149         }
150     }
151
152     @name(".parse_inner_icmp") state parse_inner_icmp {
153         packet.extract(hdr.inner_icmp);
154         transition accept;
155     }
156
157     @name(".parse_inner_tcp") state parse_inner_tcp {
158         packet.extract(hdr.inner_tcp);
159         transition accept;
160     }
161
162     @name(".parse_inner_ethernet") state parse_inner_ethernet {
163         packet.extract(hdr.inner_ethernet);
164         transition select(hdr.inner_ethernet.etherType) {
165             16w0x800: parse_inner_ipv4;
166             default: accept;
167         }
168     }
169 }
170
171 @name("mac_learn_digest") struct mac_learn_digest {
172     bit<8> in_port;
173     bit<48> mac_sa;
174 }
175
176 /************   Ingress Processing   ************/
177 control ingress(inout headers hdr, inout metadata meta, inout
    standard_metadata_t standard_metadata) {
178
179     @name(".drop")action drop() {
```

```
180         /*mark_to_drop();*/
181     }
182
183     /************   Set IF info and others   ************/
184     @name(".set_if_info") action set_if_info(bit<1> is_ext) {
185         meta.routing_metadata.mac_da = hdr.ethernet.dstAddr;
186         meta.routing_metadata.mac_sa = hdr.ethernet.srcAddr;
187         meta.routing_metadata.if_ipv4_addr = 0x7fef4800 ;
188         meta.routing_metadata.if_mac_addr = 0x010101010100;
189         meta.routing_metadata.is_ext_if = is_ext;
190     }
191
192     @name(".if_info") table if_info {
193         key = { meta.routing_metadata.if_index: exact;}
194         actions = { drop; set_if_info; }
195         default_action = drop();
196     }
197
198     /************   Process mac learn   ************/
199     @name(".generate_learn_notify") action generate_learn_notify() {
200         digest<mac_learn_digest>(32w1024, {meta.routing_metadata.if_index,
    hdr.ethernet.srcAddr });
201     }
202
203     @name(".smac") table smac {
204         key = { hdr.ethernet.srcAddr: exact; }
205         actions = { generate_learn_notify; }
206         size = 512;
207     }
208
209     /************   Tunnel control decap   ************/
210     @name(".decap_gre_inner_ipv4") action decap_gre_inner_ipv4(bit <32>
    tunnel_id) {
211         hdr.ipv4.setInvalid();
212         hdr.gre.setInvalid();
213         meta.routing_metadata.tunnel_id = tunnel_id;
214         meta.routing_metadata.dst_ipv4 = hdr.inner_ipv4.dstAddr;
215       standard_metadata.egress_port = 1;
216         meta.routing_metadata.is_ext_if = 0;
217     }
218
219     @name("decap_process_outer") table decap_process_outer {
220         actions = {decap_gre_inner_ipv4; drop; }
221         key ={ hdr.ethernet.srcAddr: exact;}
222         size = 1024;
223         default_action = drop();
224     }
```

```
225
226     /************   Nat control   ************/
227     @name(".nat_hit_int_to_ext") action nat_hit_int_to_ext(bit<32> srcAddr,
        bit<16> srcPort) {
228         meta.routing_metadata.rewrite_outer = 1w1;
229         hdr.inner_ipv4.srcAddr= srcAddr;
230         hdr.inner_tcp.srcPort = srcPort;
231     }
232
233     @name(".nat_up") table nat_up {
234         actions = { drop; nat_hit_int_to_ext; }
235         key = { hdr.inner_ipv4.srcAddr: exact;}
236         size = 1024;
237         default_action = drop();
238     }
239
240     @name(".nat_hit_ext_to_int") action nat_hit_ext_to_int(bit<32> dstAddr,
        bit<16> dstPort) {
241         meta.routing_metadata.rewrite_outer = 1w0;
242         meta.routing_metadata.dst_ipv4 = dstAddr;
243         hdr.ipv4.dstAddr = dstAddr;
244         hdr.tcp.dstPort = dstPort;
245     }
246
247     @name(".nat_dw") table nat_dw {
248         actions = { drop; nat_hit_ext_to_int;  }
249         key = {meta.routing_metadata.is_ext_if: exact; }
250         size = 1024;
251         default_action = drop();
252     }
253
254     /************   Tunnel control encap   ************/
255     @name(".ipv4_gre_rewrite") action ipv4_gre_rewrite(bit<32> gre_srcAddr)
        {
256       hdr.ethernet.setInvalid();
257       hdr.gre.setValid();
258       hdr.gre.proto = 16w0x800;
259
260       meta.meta_ipv4.version        = hdr.ipv4.version        ;
261       meta.meta_ipv4.ihl            = hdr.ipv4.ihl            ;
262       meta.meta_ipv4.diffserv       = hdr.ipv4.diffserv       ;
263       meta.meta_ipv4.totalLen       = hdr.ipv4.totalLen       ;
264       meta.meta_ipv4.identification = hdr.ipv4.identification ;
265       meta.meta_ipv4.flags          = hdr.ipv4.flags          ;
266       meta.meta_ipv4.fragOffset     = hdr.ipv4.fragOffset     ;
267       meta.meta_ipv4.ttl            = hdr.ipv4.ttl            ;
268       meta.meta_ipv4.protocol       = hdr.ipv4.protocol       ;
```

```
269         meta.meta_ipv4.hdrChecksum     = hdr.ipv4.hdrChecksum     ;
270         meta.meta_ipv4.srcAddr         = hdr.ipv4.srcAddr         ;
271         meta.meta_ipv4.dstAddr         = hdr.ipv4.dstAddr         ;
272
273         hdr.outer_ipv4.setValid();
274         hdr.outer_ipv4.srcAddr  = 0x04000001;
275         hdr.outer_ipv4.dstAddr  = gre_srcAddr;
276         hdr.outer_ipv4.protocol = 47;
277         hdr.outer_ipv4.version         = meta.meta_ipv4.version         ;
278         hdr.outer_ipv4.ihl             = meta.meta_ipv4.ihl             ;
279         hdr.outer_ipv4.diffserv        = meta.meta_ipv4.diffserv        ;
280         hdr.outer_ipv4.totalLen        = meta.meta_ipv4.totalLen        ;
281         hdr.outer_ipv4.identification = meta.meta_ipv4.identification ;
282         hdr.outer_ipv4.flags           = meta.meta_ipv4.flags           ;
283         hdr.outer_ipv4.fragOffset      = meta.meta_ipv4.fragOffset      ;
284         hdr.outer_ipv4.ttl             = meta.meta_ipv4.ttl             ;
285
286         hdr.outer_ethernet.setValid();
287         hdr.outer_ethernet.dstAddr     = 0x000000000001;
288         hdr.outer_ethernet.srcAddr     = 0x000000000002;
289         hdr.outer_ethernet.etherType   = 16w0x800;
290         standard_metadata.egress_port = 1;
291         meta.routing_metadata.dst_ipv4= hdr.outer_ipv4.dstAddr;
292         meta.routing_metadata.rewrite_outer = 0;
293     }
294
295     @name(".tunnel_encap_process_outer") table tunnel_encap_process_outer {
296         actions = {ipv4_gre_rewrite; drop; }
297         key = { hdr.ipv4.dstAddr : exact; }
298         size = 128;
299     }
300
301 /************   Forwarding IPv4   ************/
302     @name(".set_nhop") action set_nhop(bit<9> port) {
303         standard_metadata.egress_port =  port;
304     }
305
306     @name(".ipv4_up") table ipv4_up {
307         key = {meta.routing_metadata.dst_ipv4 : lpm;}
308         actions = { set_nhop; drop; }
309     }
310
311     @name(".rewrite_src_mac") action rewrite_src_mac(bit<48> src_mac) {
312         hdr.ethernet.setInvalid();
313         hdr.ethernet_decap.setValid();
314         hdr.ethernet_decap.dstAddr   = meta.routing_metadata.mac_da;
315         hdr.ethernet_decap.srcAddr   = src_mac;
```

```
316         hdr.ethernet_decap.etherType = 16w0x800;
317     }
318
319     @name(".sendout") table sendout {
320         actions = {drop; rewrite_src_mac; }
321         key = { standard_metadata.egress_port: exact; }
322         size = 512;
323     }
324
325     @name(".rewrite_src_mac_dw") action rewrite_src_mac_dw(bit<48> src_mac)
        {
326         hdr.outer_ethernet.dstAddr    = meta.routing_metadata.mac_da;
327         hdr.outer_ethernet.srcAddr    = src_mac;
328         hdr.outer_ethernet.etherType = 16w0x800;
329     }
330
331     @name(".sendout_dw") table sendout_dw {
332         actions = {drop; rewrite_src_mac_dw; }
333         key = {  standard_metadata.egress_port: exact; }
334         size = 512;
335     }
336
337 /************    APPLY    ************/
338     apply {
339         if_info.apply();
340         smac.apply();
341         /* ------ decap-------  */
342         if(hdr.ipv4.protocol== 8w47){
343            decap_process_outer.apply();
344            nat_up.apply();
345         }
346
347         if(meta.routing_metadata.is_ext_if == 1){
348            nat_dw.apply();
349            tunnel_encap_process_outer.apply();
350         }
351            ipv4_up.apply();
352
353         if (meta.routing_metadata.rewrite_outer == 1w1) {
354             sendout.apply();
355         }
356         if (meta.routing_metadata.rewrite_outer == 1w0) {
357             sendout_dw.apply();
358         }
359     }
360 }
361
```

```
362 control egress(inout headers hdr, inout metadata meta, inout
        standard_metadata_t standard_metadata) {
363     apply {
364     }
365 }
366
367 control DeparserImpl(packet_out packet, in headers hdr) {
368     apply {
369     }
370 }
371 /************   Checksum Verification   *************/
372 control verifyChecksum(inout headers hdr, inout metadata meta) {
373     apply {
374     }
375 }
376
377 control computeChecksum(inout headers hdr, inout metadata meta) {
378     apply {
379     }
380 }
381
382 V1Switch(ParserImpl(), verifyChecksum(), ingress(), egress(),
        computeChecksum(), DeparserImpl()) main;
```

Listing E.2 – BNG $P4_{16}$ code

E.2 Dependency Graphs for BNG Use Case

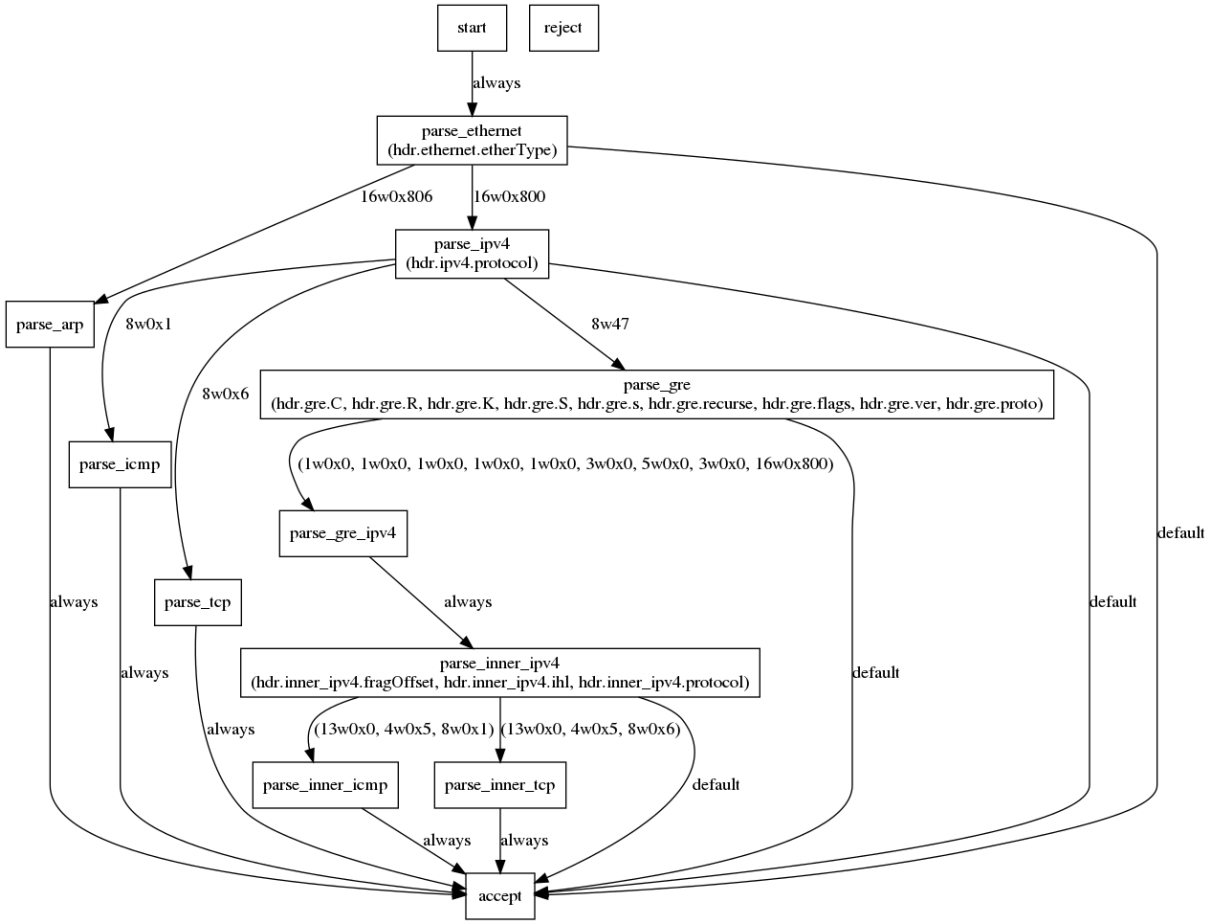


Figure 52 – BNG Parser Dependency Graph

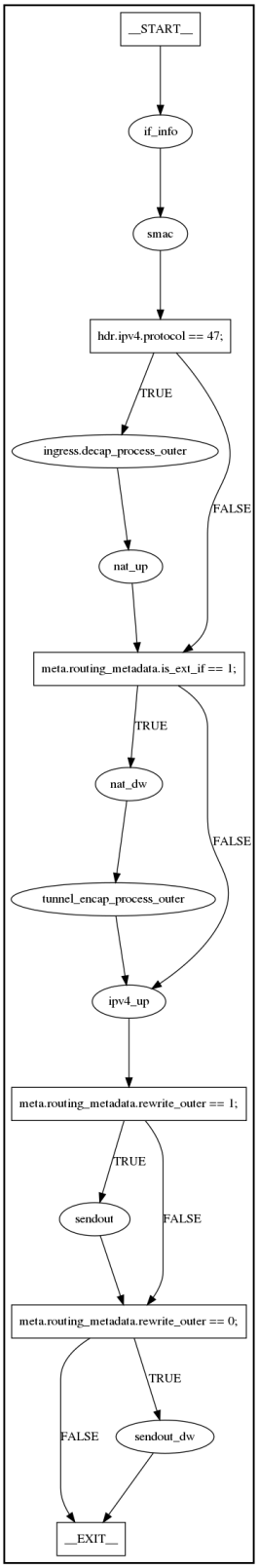


Figure 53 – BNG Table Dependency Graph