POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

FPGA-Friendly Programmable Data Plane: A Transition from P4 Semantics to Pipeline Implementations via High-Level Synthesis Approaches

MOSTAFA ABBASMOLLAEI
Département de génie informatique

Thèse présentée en vue de l'obtention du diplôme de *Philosophiæ Doctor*
Génie informatique

Décembre 2025

© Mostafa Abbasmollaei, 2025.

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Cette thèse intitulée :

FPGA-Friendly Programmable Data Plane: A Transition from P4 Semantics to Pipeline Implementations via High-Level Synthesis Approaches

présentée par **Mostafa ABBASMOLLAEI**
en vue de l'obtention du diplôme de *Philosophiæ Doctor*
a été dûment acceptée par le jury d'examen constitué de :

Alejandro QUINTERO, président
Tarek OULD-BACHIR, membre et directeur de recherche
Yvon SAVARIA, membre et codirecteur de recherche
Jean Pierre DAVID, membre
Mohamed-Faten ZHANI, membre externe

DEDICATION

Perhaps no one ever noticed you, nor the countless efforts you made for me on this journey. You were there when I built my very first prototype, and you stayed awake tirelessly for more than a year, giving me access to my hardware through every challenge, in cold and heat alike. Yes, I am talking to you: my little Raspberry Pi.

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Tarek Ould-Bachir, for his invaluable guidance, patience, and continuous support throughout my PhD journey. His expertise, insightful feedback, and encouragement have shaped not only the direction of this research but also my growth as a researcher. I am truly fortunate to have had the opportunity to learn from his knowledge and mentorship.

I am also sincerely grateful to my co-supervisor, Yvon Savaria, for his constructive advice, thoughtful suggestions, and constant availability whenever I needed help. His careful attention to detail and practical insights have been essential in refining my work and advancing this thesis.

My heartfelt thanks go to my spouse, Fahimeh Hajizadeh, who has been not only my greatest support but also the best friend and companion in my life. She has stood by me through every up and down, making sacrifices to allow me the space and opportunity to grow. Her unwavering motivation, patience, and constant encouragement have carried me through the most challenging moments of this PhD. Her belief in me has been an endless source of strength and inspiration, without which this journey would have been far more difficult.

I would also like to express my deepest appreciation to my family, who have always been with me in spirit despite the physical distance. To my mother, thank you for your unconditional love, encouragement, and sacrifices, which have given me the strength to move forward. My late father's greatest wish was to see me earn a PhD, and his memory has been a constant source of motivation throughout this journey. I dedicate this achievement to his soul with profound gratitude, as this accomplishment belongs as much to him as it does to me.

Finally, I would like to thank the jury members for generously giving their time to evaluate my thesis. Their insightful comments, constructive feedback, and thoughtful questions have greatly enriched this work and helped me strengthen its contributions.

To all of you, I am profoundly grateful.

RÉSUMÉ

L'évolution constante des réseaux à haut débit exige des plans de données de plus en plus programmables, capables de s'adapter à une grande diversité de protocoles et d'applications sans compromettre les performances. Les technologies matérielles émergentes dans le domaine des réseaux de données telles que les Field-Programmable Gate Arrays (FPGAs) offrent une solution particulièrement prometteuse en combinant reconfigurabilité, parallélisme massif et débit déterministe. Cependant, malgré plusieurs tentatives visant à implémenter les programmes P4 sur des plateformes FPGA, les solutions existantes demeurent limitées en termes de portabilité, de flexibilité et d'évolutivité. La plupart nécessitent des modifications manuelles, des chaînes d'outils spécifiques à un fournisseur ou des composants propriétaires, rendant le déploiement et la personnalisation complexes et sujets à des erreurs.

Cette thèse propose une méthodologie complète et automatisée pour combler le fossé entre la programmabilité offerte par P4 et la réalisation matérielle sur FPGA, en offrant un flux de travail intégré qui élimine les difficultés de conversion et de déploiement manuel. Au cœur de cette méthodologie se trouve P4THLS, un cadre de mise en oeuvre basé sur la synthèse de haut niveau (HLS) et conçu sous forme de modèles, servant de fondation au noyau de traitement des paquets. P4THLS traduit automatiquement les programmes P4 en modules HLS synthétisables, générant tous les composants principaux d'un plan de données programmable, y compris le parseur, les tables de correspondance et d'action (match-action) et le déparseur, tout en prenant en charge plusieurs types de mémoires embarquées pour le stockage et l'accès efficaces aux entrées de table.

Afin de faciliter l'implémentation de tables de correspondance-action performantes, ce travail introduit HLSCAM, une bibliothèque purement HLS qui explore l'espace de conception des mémoires associatives binaires et ternaires (BCAM et TCAM). HLSCAM permet la création de structures de recherche efficaces, indépendantes du fournisseur et facilement extensibles, intégrées de manière fluide dans la chaîne de traitement P4THLS. Le cadre a ensuite été enrichi par la prise en charge d'objets à états (tels que les registres, compteurs et compteurs de débit) et la propagation de métadonnées, permettant la mise en œuvre de fonctions avancées de traitement de paquets comme la télémétrie, le contrôle de flux et la surveillance du trafic. Ces extensions offrent une meilleure visibilité sur le pipeline interne et permettent aux applications complexes de fonctionner à débit de ligne sans dépendre d'interactions avec le plan de contrôle externe. La validation expérimentale a été effectuée sur une plateforme FPGA AMD Alveo U280, sous un trafic réaliste généré à l'aide des cartes réseau Intel XXV710 et

du générateur de trafic TRex.

Dans son ensemble, cette thèse établit P4THLS comme une fondation unifiée et extensible pour le déploiement d'applications de plan de données définies en P4 sur des plateformes FPGA. En combinant automatisation, abstraction et conception orientée performance, ce travail ouvre la voie à un nouveau niveau de programmabilité, capacité d'extension et efficacité matérielle, jetant les bases des systèmes réseaux intelligents et auto-adaptatifs de prochaine génération basés sur FPGA.

ABSTRACT

The continuous evolution of high-speed networking demands increasingly programmable data planes capable of adapting to diverse protocols and applications without sacrificing performance. Emerging hardware technologies such as Field-Programmable Gate Arrays (FPGAs) offer a compelling solution combining reconfigurability, massive parallelism, and deterministic throughput. However, despite several attempts to map P4 programs onto FPGA platforms, existing solutions remain limited in portability, flexibility, and scalability. Most require manual modifications, vendor-specific toolchains, or proprietary components, making deployment and customization complex and error-prone.

This dissertation proposes a comprehensive and automated methodology to bridge the gap between P4 programmability and FPGA-based realization, offering a seamless workflow that eliminates costly manual conversion and deployment. At the heart of this methodology lies P4THLS, a templated High-Level Synthesis (HLS) framework that serves as the foundation of the packet processing kernel. P4THLS automatically translates P4 programs into synthesizable HLS modules, generating all major components of a programmable data plane, including the parser, match-action tables, and deparser, with support for multiple on-chip memory types to store and access table entries efficiently.

This work introduces HLSCAM, a pure-HLS library that explores the design space of binary and ternary content-addressable memories (BCAMs and TCAMs) to facilitate the implementation of high-performance match-action tables. HLSCAM enables the development of resource-efficient, vendor-agnostic, and easily scalable lookup structures that integrate seamlessly into the P4THLS pipeline. The framework was further enhanced with stateful object support, such as registers, counters, and meters, and metadata propagation, enabling advanced packet processing functions like telemetry, flow control, and traffic monitoring. These extensions provide greater visibility into the internal pipeline, allowing complex applications to operate at line rate without relying on external control-plane interactions. Experimental validation was performed on an AMD Alveo U280 FPGA platform under realistic traffic using Intel XXV710 network adapters and the TRex traffic generator.

Overall, this dissertation establishes P4THLS as a unified and extensible foundation for deploying P4-defined data-plane applications on FPGA platforms. By combining automation, abstraction, and performance-oriented design, it enables a new level of programmability, scalability, and hardware efficiency, paving the way for next-generation intelligent and self-adaptive FPGA-based network systems.

TABLE OF CONTENTS

DEDICATION iii

ACKNOWLEDGEMENTS iv

RÉSUMÉ v

ABSTRACT vii

TABLE OF CONTENTS viii

LIST OF TABLES xiii

LIST OF FIGURES xiv

LIST OF SYMBOLS AND ACRONYMS xvi

CHAPTER 1 INTRODUCTION 1
- 1.1 Context 1
  - 1.1.1 Software-Defined Networking 1
  - 1.1.2 Data Plane Programmability 2
  - 1.1.3 P4 Language 2
- 1.2 Problem Definition 3
- 1.3 Research Objectives 4
- 1.4 Contributions 5
- 1.5 Thesis Outline 6

CHAPTER 2 BACKGROUND 8
- 2.1 P4 Specifications and Capabilities 8
  - 2.1.1 Headers and Metadata 8
  - 2.1.2 Parser 8
  - 2.1.3 Match-Action Tables (MATs) 8
  - 2.1.4 Control Flow 9
  - 2.1.5 Deparser 9
- 2.2 P4 Limitations and Design Challenges 9
- 2.3 Programmable Data Plane Targets 10
  - 2.3.1 Software (CPU-based) 11

2.3.2 ASIC 11
2.3.3 NPU 12
2.3.4 FPGAs 12

CHAPTER 3 LITERATURE REVIEW 14
3.1 Introduction 14
3.2 FPGA-accelerated Programmable Data Plane 14
3.3 CAMs on FPGA 18
3.4 P4 Conversion 20
3.5 P4 to FPGA Transition 21

CHAPTER 4 CONTENT ORGANIZATION AND RESEARCH APPROACH . . . 24
4.1 Content Organization 24
4.2 Research Methodology 25
4.2.1 Conceptual and Analytical Phase 25
4.2.2 Framework Design and Development 25
4.2.3 Framework Extension and Functional Enrichment 26
4.2.4 Experimental Validation 27

CHAPTER 5 ARTICLE 1: NORMAL AND RESILIENT MODE FPGA-BASED ACCESS GATEWAY FUNCTION THROUGH P4-GENERATED RTL 28
5.1 Introduction 28
5.2 Background and Related Work 30
5.2.1 Access Gateway Function (AGF) 30
5.2.2 FPGA-based P4 5G Core Functions 30
5.2.3 P4-based HDL 31
5.3 Proposed Architecture 33
5.3.1 FPGA-based Architecture for the AGF 33
5.3.2 Implementing P4-based Pipelines on FPGA 33
5.3.3 Resilient Operating Mode 35
5.4 Evaluation 39
5.4.1 Experimental Setup 39
5.4.2 Implementation Results 40
5.4.3 Scalability Results 41
5.5 Conclusion 42

CHAPTER 6 ARTICLE 2: HLSCAM: FINE-TUNED HLS-BASED CONTENT ADDRESSABLE MEMORY IMPLEMENTATION FOR PACKET PROCESSING ON FPGA . . . 43
6.1 Introduction . . . 43
6.2 Literature Review . . . 46
6.3 Potential Network Applications . . . 48
6.3.1 Packet Classification and Filtering . . . 48
6.3.2 Firewall and Intrusion Detection Systems (IDS) . . . 48
6.3.3 In-Band Network Telemetry (INT) . . . 48
6.3.4 Network Address Translation (NAT) . . . 49
6.3.5 Time-Sensitive Networking (TSN) . . . 49
6.3.6 User Management in 5G Core . . . 49
6.4 CAM Architectures, Implementations, and Trade-Offs . . . 49
6.4.1 Design Perspectives . . . 50
6.4.2 BCAM . . . 53
6.4.3 TCAM . . . 54
6.4.4 Memory Allocation . . . 55
6.5 Evaluation . . . 59
6.5.1 Experimental Setup . . . 59
6.5.2 Design Space Exploration . . . 59
6.5.3 BCAM Versus TCAM . . . 60
6.5.4 High-Speed (HS) Optimization Mode Variations . . . 64
6.5.5 Comparison . . . 65
6.6 Discussion . . . 68
6.7 Conclusions . . . 69

CHAPTER 7 ARTICLE 3: P4THLS: A TEMPLATED HLS FRAMEWORK TO AUTOMATE EFFICIENT MAPPING OF P4 DATA-PLANE APPLICATIONS TO FPGAS . . . 71
7.1 Introduction . . . 71
7.2 Literature Review and Background . . . 73
7.2.1 P4 and Data Plane Packet Processing . . . 73
7.2.2 FPGA-Based Implementations for P4 Codes . . . 73
7.2.3 Templated Architecture . . . 75
7.3 P4THLS: Architecture and Implementation . . . 75
7.3.1 Workflow from P4 to HLS . . . 75

7.3.2 Packet Processing Pipeline 78
7.3.3 P4THLS Parameterization and Flexibility 82
7.3.4 Unified Memory Interface 82
7.3.5 P4THLS Example: UDP Access Control List 84
7.3.6 Dataflow and Performance Characteristics 84
7.4 Evaluation and Performance Analysis 86
7.4.1 Analysis of Different CAM Parameters 86
7.4.2 Variable Bus Width 87
7.4.3 Qualitative Comparative Analysis 89
7.4.4 Empirical Comparative Analysis 90
7.4.5 Discussion 98
7.5 Future Work 99
7.6 Conclusion 99

CHAPTER 8 ARTICLE 4: A STATEFUL EXTENSION TO P4THLS FOR ADVANCED TELEMETRY AND FLOW CONTROL 101
8.1 Introduction 101
8.2 Background and Related Work 103
8.2.1 P4 Language and Data Plane Packet Processing 103
8.2.2 Stateful Processing in P4 Data Planes 104
8.2.3 Meter Algorithms 105
8.2.4 In-band Network Telemetry 105
8.2.5 Network Flow Control 106
8.3 Proposed Architecture, Automation, and Implementation 106
8.3.1 Fundamental Packet Processing Architecture 107
8.3.2 Stateful Object Support 110
8.3.3 Packet Processing Kernel Integration and Automation 115
8.3.4 Architecture for In-band Network Telemetry 116
8.3.5 Architecture for Flow Control 117
8.4 Evaluation 119
8.4.1 Experimental Setup 119
8.4.2 Comparison with Baseline 119
8.4.3 Comparative Evaluation with Existing Work 121
8.5 Conclusion 123

CHAPTER 9 GENERAL DISCUSSION 125
9.1 Interpretation of Results 125

9.2 Comparison to Existing Literature 126
9.3 Addressing Research Objectives, Significance, and Implications 128

CHAPTER 10 CONCLUSION 130
10.1 Summary of Contributions 130
10.2 Summary of Works 132
10.3 Limitations 134
10.4 Future Directions 135

REFERENCES 137

LIST OF TABLES

Table 2.1 The difference between the P4 targets (inspired from [1]) 10
Table 5.1 FPGA resource utilization for different numbers of supported user flows 40
Table 5.2 The comparison of latency and the number of user flows that each work supports 41
Table 6.1 Resource Utilization and Performance Analysis of HLSCAM BCAM in the BF, BL, and HS modes. The value width is fixed to 8-bit. 61
Table 6.2 Resource Utilization and Performance Analysis of HLSCAM TCAM in the BF, BL, and HS modes. A TCAM entry holds key and mask portions having the same length. The value width is fixed to 8-bit. . . 62
Table 6.3 Different FPGA parameters 66
Table 6.4 Resource Utilization and Throughput Comparison with Existing FPGA-based CAM Architectures 67
Table 7.1 Existing P4-to-FPGA frameworks: features and limitations 76
Table 7.2 TP4THLS Parameters and Affected Measures 83
Table 7.3 Performance results for different match kinds used in match-action tables (BL optimization is used for TCAM, STCAM, and BCAM with PF = 32) 87
Table 7.4 Comparison of P4THLS, VitisNetP4, and P4-to-FPGA 89
Table 7.5 Different FPGA parameters. 92
Table 7.6 Comparison of resource utilization, latency, and throughput between P4THLS and P4-to-FPGA IP for three P4 Applications ($f_{clk} = 250$ MHz) 93
Table 7.7 Comparison of resource utilization and latency between P4THLS and AMD Xilinx VitisNetP4 IP for two P4 Applications with varying parameters ($f_{clk} = 250$ MHz) 96
Table 8.1 Comparison of the baseline P4THLS with different enriched stateful options (IC: Indirect Counter, IM: Indirect Meter) 120
Table 8.2 Comparison of FPGA resource utilization for a stateful network use case 123

LIST OF FIGURES

Figure 5.1 5G core network schematic, inspired from [2] 31
Figure 5.2 The data flow of the AGF application, adopted from [3]. 32
Figure 5.3 Experimental setup of the proposed FPGA implemented AGF architecture 34
Figure 5.4 The synthesized schematic of a P4 IP instance within the AGF implementation 34
Figure 5.5 The EFSM diagram for detecting and mitigating volumetric DDoS attacks 37
Figure 6.1 The use of CAM tables in L2 switching. 44
Figure 6.2 The hardware structure of CAM lookup in the BF mode on FPGA. . 50
Figure 6.3 The hardware structure of CAM lookup in the BL mode on FPGA. . 51
Figure 6.4 The hardware structure of CAM lookup in the HS mode on FPGA. . 53
Figure 6.5 The implemented structures of BCAM and TCAM. 54
Figure 6.6 The TCAM search entry logic. 55
Figure 6.7 Memory allocation for the proposed BCAM in the BF mode with double memory ports and depth = 64, entry width = 25 bits (16-bit key, 8-bit value, 1-bit valid). 56
Figure 6.8 Memory allocation for the proposed BCAM in the BL mode with $PF = 16$, depth = 64, entry width = 25 bits (16-bit key, 8-bit value, 1-bit valid). 58
Figure 6.9 Comparison of the proposed BCAM and TCAM in the mode BF for 128 and 256 CAM depth (128-bit key, 8-bit value, 1-bit valid). 63
Figure 6.10 Comparison of the proposed BCAM and TCAM in the mode BL for 128 and 256 CAM depth (128-bit key, 8-bit value, 1-bit valid). 63
Figure 6.11 Comparison of the proposed BCAM and TCAM in the mode HS for 128 and 256 CAM depth (128-bit key, 8-bit value, 1-bit valid). 64
Figure 6.12 Comparison of the HS and HS-H modes for the proposed TCAM (128-bit key, 8-bit value, 1-bit valid). 65
Figure 7.1 PISA architecture 74
Figure 7.2 P4THLS Automation Workflow 76
Figure 7.3 The template architecture of the HLS project generated by P4THLS . 77
Figure 7.4 Steps to call the `apply()` in the table handler object 79

Figure 7.5 Example: An overview of the C++ generated kernel of a UDP Access Control List using the P4THLS 83
Figure 7.6 A latency-aware schematic of the P4THLS kernel's pipeline 85
Figure 7.7 Impact of bus width on latency considering various packet sizes for a UDP Firewall application (f_{clk} = 250 MHz) 88
Figure 7.8 Experimental Setup - controller, FPGA block design, and traffic generation setup 88
Figure 7.9 Packet flow between the QSFP28 cage and the P4THLS packet processing kernel 90
Figure 8.1 P4THLS packet processing pipeline [4] (extended with the proposed features) and inter-module interfaces. `Ft` and `Tm` denote the Fetch and Transmit blocks. 108
Figure 8.2 Meter algorithm using two colors and a single rate, assuming a 250 MHz clock frequency. 114
Figure 8.3 A simplified automation workflow: P4 to FPGA deployment (inspired by [4]) 115
Figure 8.4 Custom header attachment for INT - PsTs: Parser Timestamp, IgDTs: Ingress Differential Timestamp, EgDTs: Egress Differential Timestamp, EnqQ: Input Queue Depth 116
Figure 8.5 64-byte packet format for INT including custom headers - PsTs: Parser Timestamp, IgDTs: Ingress Differential Timestamp, EgDTs: Egress Differential Timestamp, EnqQ: Input Queue Depth 117
Figure 8.6 Meter mechanism to control the flow rate 118
Figure 8.7 The achieved rate and absolute relative error to the target rate compared to Wang et al. [5]. 122

LIST OF SYMBOLS AND ACRONYMS

AGF	Access Gateway Function
AQM	Active Queue Management
ASIC	Application Specific Integrated Circuit
BCAM	Binary CAM
BRAM	Block RAM
CAM	Content Addressable Memory
CBS	Committed Burst Size
CIR	Committed Information Rate
DDR	Double Data Rate memory
EBS	Excess Burst Size
EFSM	Extended Finite State Machines
FF	Flip-Flop
FPGA	Field-Programmable Gate Arrays
HBM	High-Bandwidth Memory
HBCAM	Hash-based Binary CAM
HDL	Hardware Description Language
INT	In-band Network Telemetry
LPM	Long-Prefix Match
LUT	Look-Up Table
MAT	Match-Action Table
ML	Machine Learning
NFV	Network Functions Virtualization
NIC	Network Interface Card
NPU	Network Processor Unit
P4	Programming Protocol-Independent Packet Processors
PIR	Peak Information Rate
PHV	Packet Header Vector
PISA	Protocol Independent Switch Architecture
PPPoE	Point-to-Point Protocol over Ethernet
QoS	Quality-of-Service
QSFP	Quad Small Form-factor Pluggable
RMT	Reconfigurable Match-Action Table
SDN	Software-Defined Networking

srTCM	Single-Rate Three-Color Marker
STCAM	Semi-Ternary CAM
TCAM	Ternary CAM
trTCM	Two-Rate Three-Color Marker
UPF	User Plane Function
URAM	Ultra RAM
VLAN	Virtual Local Area Network
VNF	Virtual Network Function

CHAPTER 1 INTRODUCTION

1.1 Context

Networking has experienced significant progress in recent years, impacting various fields such as social networks, cloud services, the Internet of Things (IoT), and security. Social networks have shifted worldwide communication, while cloud services have revolutionized company processes. Networking and IoT make device connectivity and data exchange possible. Innovative solutions have been introduced to safeguard networks against different kinds of cyber threats. The important and effective changes in networking are emerging Software-Defined Networking (SDN) and programmable data planes, which are game-changing innovations that provide centralized management and customization to boost network management. These advances continue to shape and redefine the networking landscape.

1.1.1 Software-Defined Networking

Software-Defined Networking is an approach that separates the control plane from the data plane, providing centralized control and programmability. It offers several benefits such as flexibility, scalability, network virtualization, enhanced security, and integration with emerging technologies. SDN's flexibility and programmability allow for rapid provisioning, automation, and customization of network services. It simplifies network management and enables adaptation to changing business needs. The separation of the control plane and data plane also improves scalability and resource utilization. Network virtualization is another advantage of SDN, allowing multiple logical networks to coexist on a shared physical infrastructure. This facilitates deploying cloud services, multi-tenancy, and isolated network segments with specific policies.

In traditional networking, both the control plane and data plane are controlled entirely by the device vendors. However, SDN introduces a clear separation between the control plane and the data plane, consolidating the control plane under a centralized controller. Unlike traditional devices, the controller in SDN is implemented in software and is controlled by the network owner. The controller performs computations to generate tables used by individual switches and distributes them through a well-defined API, such as OpenFlow [6].

While SDN allows for customization of the control plane, it is constrained by OpenFlow's specifications and the data plane's fixed functionality. In other words, SDN enables the network owner to have centralized control over multiple data planes, with the controller

making decisions and distributing instructions to the switches. The separation of the control and data planes allows for greater flexibility and programmability in network management. However, it should be noted that the extent of customization in SDN is confined to the specifications defined by OpenFlow and the capabilities of the underlying hardware in the data plane.

1.1.2 Data Plane Programmability

While SDN decreased network complexity and accelerated control plane innovation at the speed of software development, management of real packet processing operations remained the onus of network vendors. Additionally, the design cycle of switches implemented as application specific integrated circuits (ASICs) is a proprietary process that usually takes years to complete. This procedure contrasts with the software industry's agility [6, 7]. The idea of programmable data planes was proposed to avoid the operators having to wait for conventional networking hardware. In fact, the device's programming interface enables operators to rapidly create and release new features [8, 9].

1.1.3 P4 Language

The language for Programming Protocol-independent Packet Processors (P4) is currently the most widespread programming language for data plane programming. It was introduced in 2014 for data plane programmability in networking devices. P4 provides a high-level abstraction that allows network operators to define and customize the behavior of the data plane, which is responsible for packet forwarding and processing in network devices such as switches and routers. P4 focuses on the data plane, which is responsible for packet forwarding and processing. By using P4, network operators can define custom packet processing behaviors, including parsing, matching, and modifying packet headers. This level of programmability enables the creation of innovative networking functionalities and protocols [10].

One key advantage of P4 is its protocol independence. It allows network operators to define how protocols are processed and how packets are handled, regardless of the underlying hardware or existing protocols. This flexibility enables rapid experimentation and deployment of new protocols and network functions.

P4-based programmable switches and devices offer several benefits. They provide a standardized and vendor-agnostic approach to data plane programmability, enabling interoperability and reducing vendor lock-in. With P4, network operators have the freedom to define their own forwarding behaviors and protocols, leading to more flexible and customizable network

architectures. P4 also aligns with the concepts of SDN and Network Functions Virtualization (NFV). It enables the separation of the control plane and data plane, allowing for centralized control and management of network devices. This integration of P4 with SDN and NFV allows for dynamic and programmable networks, where network functions can be virtualized, orchestrated, and automated.

The adoption of P4 is growing in the networking industry as it provides a powerful tool for network innovation and advancement. It fosters collaboration and encourages the development of new protocols, services, and applications. By leveraging the programmability offered by P4, network operators can create more efficient, agile, and adaptable networks that can meet the evolving demands of modern applications and services.

1.2 Problem Definition

P4 has emerged as a high-level language for defining the behavior of data planes, providing a flexible means to customize packet processing. By abstracting forwarding logic and protocol parsing, P4 enables researchers and engineers to prototype new network designs, validate ideas, and deploy innovative policies without relying on the slow cycle of fabricating fixed-function equipment. This capability has placed P4 as a key enabler for accelerating network innovation, particularly in domains such as software-defined networking, network function virtualization, and programmable switches.

At the same time, FPGAs have become increasingly prevalent across computing and networking platforms, including servers, accelerators, and network interface cards (NICs). Their high-speed I/O interfaces, fine-grained parallelism, and reconfigurable logic make them especially well-suited for networking workloads. Unlike CPUs or GPUs, which may face bottlenecks when processing packets at line rate, FPGAs can sustain multi-terabit throughput while remaining reprogrammable to accommodate evolving protocols and policies. These properties have already led to widespread adoption of FPGA in datacenter accelerators (as exemplified by the Microsoft's Catapult project), SmartNICs, and edge computing devices.

Therefore, deploying P4 specifications directly onto FPGAs represents a promising opportunity to combine P4's programmability with the raw efficiency of FPGA hardware. For example, a custom firewall or packet classification system written in P4 could be quickly evaluated and deployed on an FPGA to achieve both flexibility in defining rules and line-rate performance in enforcing them. Similarly, operators could test new tunneling protocols or customized load balancers by rapidly prototyping them in P4 and implementing them on FPGA-based NICs without extensive hardware design expertise.

However, despite this promise, existing P4-to-FPGA solutions remain inadequate for practical adoption. Several notable challenges persist:

1. Limited Portability: Some frameworks are tied to specific FPGA vendors or boards, reducing their applicability across diverse platforms commonly used in datacenters or edge networks.
2. Outdated or Incomplete Toolchains: Existing compiler backends for FPGA often lag behind current P4 standards or do not support the full set of P4 constructs, limiting the range of applications that can be deployed.
3. Dependency on Third-Party IPs: Many solutions rely on external intellectual property blocks or proprietary tools to implement core functionalities such as match-action tables or memory access, introducing complexity and reducing transparency.
4. Complex Integration: Even if a P4 design is synthesized to hardware, additional effort is required to integrate it with FPGA toolchains, runtime environments, and host interfaces, which is often left unaddressed.
5. Performance vs. Flexibility Tradeoffs: Approaches that focus on programmability often suffer from resource usage or latency inefficiencies, while highly optimized designs compromise generality and ease of customization.

These challenges make mapping P4 programs to FPGA implementations cumbersome and error-prone, slowing down prototyping cycles and limiting P4's potential in real-world FPGA-based networking. Therefore, there is a strong need for a systematic, high-level, and platform-agnostic methodology that bridges the gap between P4 programmability and FPGA efficiency. A solution that eliminates the reliance on third-party IPs, automates the generation of FPGA-ready designs, and supports practical toolchain integration would significantly enhance the productivity of network prototyping, evaluation, and deployment.

1.3 Research Objectives

The main objective is to enable FPGA-based platforms to act as programmable data planes for P4 applications with flexibility, efficiency, and performance control. To achieve this, the research first establishes a methodology that bridges the gap between existing programmable architectures and FPGA-based platforms. Next, it focuses on designing and implementing a flexible, high-performance FPGA data plane that directly supports P4 programs, while

automating the deployment for end users. The proposed solution is expected to support the main elements of the P4 language, including its blocks, objects, and capabilities. Finally, it integrates mechanisms for monitoring, measuring, and evaluating the deployed data plane from the timing, performance, and validity perspectives. Based on these considerations, the research pursues the following specific objectives:

Specific Objectives:

1. Propose an applicable and flexible methodology that positions FPGA platforms as a straightforward solution for deploying network applications, by identifying the gap between existing programmable architectures and FPGA-based platforms, and analyzing their challenges, limitations, and available capabilities.

2. Design and implement a high-bandwidth, efficient, flexible, and configurable FPGA-based data plane capable of accepting P4 code, including parser, match-action tables, and deparser blocks, while supporting a relevant control plane interface.

3. Propose and develop an approach for enabling performance measurement and control across the FPGA-based data plane to evaluate key design criteria.

1.4 Contributions

This dissertation advances the state of the art through several scientific and technical contributions that encompass methodological foundations, the design and realization of specialized FPGA-based data plane architectures, and their systematic experimental evaluation. The principal contributions are outlined as follows:

1. Templated CAM Implementations [C1]: A parameterized HLS-based architectures was developed for BCAMs and TCAMs, supporting arbitrary table sizes to process data plane packets. The design space was explored across memory depths, key widths, and optimization strategies, allowing for a trade-off between resource usage and latency while achieving high operating frequencies and low look-up latency.

2. Flexible and Unified Data-Plane Architecture [C2]: An FPGA pipeline architecture that contains templated FPGA pipelines was proposed. It supports variable bus widths, high-bandwidth AXI data streams, and a unified memory interface that can be adapted to different memory types depending on the application requirements. This provides a scalable and reusable foundation for FPGA-based packet processing.

3. Automated P4-to-HLS Workflow [C3]: An automated toolchain that translates P4 programs into human-readable HLS data structures was designed and implemented. It generates parsers, match-action, and deparser blocks, and produces compatible drivers for control plane management. The framework is provided as an open-source to support adaptation and extension by the research community.

4. Integrated Hardware and Experimental Framework [C4]: A practical FPGA-based test environment was built. It consists of a high-end accelerator card as the data-plane kernel, NICs for traffic generation and monitoring, and host PCs for control plane operations. This setup integrates multiple development tools at different design levels.

5. Metadata and Stateful Object Support [C5]: The FPGA pipeline was extended with structured metadata fields (e.g., timestamps, packet length, queue depth) integrating stateful objects such as registers, counters, and meters. These enhancements broaden the applicability of the solution to advanced use cases such as in-band network telemetry (INT) and flow control.

This thesis resulted in the publication of several peer-reviewed conference and journal articles, with one additional work currently under review. The relationship between the thesis contributions (C1–C5) and the published works is summarized as follows:

1. Article 1: Normal and Resilient Mode FPGA-Based Access Gateway Function Through P4-Generated RTL [11] [supports C4]

2. Article 2: HLSCAM: Fine-Tuned HLS-Based Content Addressable Memory Implementation for Packet Processing on FPGA [12] [supports C1]

3. Article 3: P4THLS: A Templated HLS Framework to Automate Efficient Mapping of P4 Data-Plane Applications to FPGAs [4] [supports C2, C3, and C4]

4. Article 4: A Stateful Extension to HLS-Based Data Plane Processing for Advanced Telemetry and Flow Control (submitted) [supports C5]

1.5 Thesis Outline

This dissertation is organized into ten chapters including this introduction chapter.

Chapter 2 provides the necessary background, beginning with an overview of P4 specifications and capabilities, covering its main components including headers, metadata, parsers,

match-action tables, control flow, and deparsers. It then discusses the limitations and design challenges of P4, followed by an overview of programmable data plane targets across software, ASICs, NPUs, and FPGAs.

Chapter 3 presents the literature review, which examines prior work in FPGA-accelerated programmable data planes, the design and optimization of CAMs on FPGAs, approaches to P4 conversion, and frameworks for enabling P4-to-FPGA transitions.

Chapter 4 describes the content organization and overall research approach adopted in this thesis, positioning the subsequent chapters within a coherent methodology.

Chapter 5 contains the first article, which introduces an FPGA-based access gateway function for normal and resilient modes, realized through P4-generated RTL.

Chapter 6 presents the second article, detailing HLSCAM, a fine-tuned HLS-based content-addressable memory implementation tailored for packet processing on FPGAs.

Chapter 7 covers the third article, P4THLS, which proposes a templated HLS framework that automates the efficient mapping of P4 data-plane applications onto FPGAs.

Chapter 8 concludes with the fourth article, which extends the HLS-based data plane with metadata and stateful functionality to support advanced telemetry and flow control.

Chapter 9 provides a general discussion that integrates the findings from all previous chapters. It highlights the connections between individual studies, discusses the overall implications of the proposed methodologies, and reflects on how the combined contributions advance the field of FPGA-based programmable data planes.

Chapter 10 concludes the thesis by summarizing the key outcomes, emphasizing the main contributions, and outlining potential directions for future research.

CHAPTER 2 BACKGROUND

2.1 P4 Specifications and Capabilities

The P4 language is designed to describe the behavior of the packet processing data plane in a flexible and protocol-independent manner. A typical P4 program is structured around five fundamental components, each representing a stage in the data plane pipeline [8]. These components are: *headers and metadata*, *parser*, *match-action tables*, *control flow*, and *deparser*. The following subsections describe each of these in more detail.

2.1.1 Headers and Metadata

Headers specify the layout and fields of packet protocols (e.g., Ethernet, IPv4, TCP), while metadata represents additional information passed between processing stages. Metadata can include both standard fields (e.g., ingress port, packet length) and custom user-defined fields that assist in implementing specific processing logic. Together, headers and metadata form the context that guides packet parsing, classification, and modification.

2.1.2 Parser

The parser is responsible for extracting headers from the incoming packet stream. In P4, it is expressed as a finite state machine (FSM), where each state defines the extraction of a single header structure and the transition to the next state depends on field values (e.g., the EtherType field determines whether to parse an IPv4 or IPv6 header). This FSM-based abstraction provides flexibility to define complex protocol stacks and conditional parsing behaviors.

2.1.3 Match-Action Tables (MATs)

Match-action tables represent the core of packet processing in P4. Each table specifies the fields to be matched and the associated actions to be executed. Table entries are populated dynamically by the control plane at runtime, allowing for fast reconfiguration of forwarding behavior. Actions are small code blocks capable of reading and modifying headers and metadata. They are typically invoked by MATs and resemble functions in traditional programming languages, except they do not return values. Instead, they directly manipulate the packet state.

A key strength of MATs is their support for different *types of matching*:

- Exact Match: The packet field must match the table entry value exactly. This is commonly used for tasks such as mapping VLAN IDs or protocol identifiers.
- Longest Prefix Match (LPM): Matches based on prefixes of variable length, where the longest prefix is selected when multiple entries match. LPM is frequently used in IP routing tables.
- Ternary Match: Allows "don't care" bits (wildcards) in the match fields, providing flexibility for implementing functions such as access control lists (ACLs), classification rules, or policy enforcement.

By supporting these match types, MATs enable a wide spectrum of network functions, from deterministic lookups (exact match), to scalable forwarding (LPM), and fine-grained policy enforcement (ternary).

2.1.4 Control Flow

The flow controller, often referred to as the body of a control block, specifies the order and logic in which tables, actions, and externs are applied. It includes a mandatory `apply()` method, which defines the execution sequence of resources and thus determines the packet processing pipeline. Through control blocks, P4 supports conditional execution, modularity, and the composition of complex pipelines.

2.1.5 Deparser

After processing, the deparser reconstructs the packet into a byte stream for transmission. It serializes headers and payloads, assembling them in the correct order to reflect modifications made during processing. The deparser ensures packets are properly formatted before they are forwarded to the appropriate destination, which may include another data plane pipeline, a CPU interface, or an external port.

2.2 P4 Limitations and Design Challenges

P4 does not support floating point numbers. In addition, due to the lack of loop constructs, P4 is unable to process the packet payload efficiently. Moreover, there is no support for

pointers, references, and dynamic memory allocation. These language limitations make it difficult to implement efficient algorithms that require deep packet inspection.

Despite the growing interest in the programmability of the data plane, its widespread adoption depends on advances in several key areas, including data plane security, the availability of P4 programmable switches, migration costs, P4 design issues, packet processing speed, and the implementation of stateful network functions. There are some security challenges in the P4-Programmable data planes. Firstly, the software is more prone to errors than the hardware. Moreover, data plane forwarding behavior is defined by the end user, who is normally less experienced and careful as compared to the vendor. Secondly, an attacker can exploit the programmability feature to modify the forwarding behavior of the device, which creates new attack vectors. Therefore, assertions to be added to the programs and verification should be performed to improve the security of the programmable data plane.

2.3 Programmable Data Plane Targets

P4 programs can execute on a range of data-plane targets that we group into four classes: software targets on general-purpose CPUs, network processing units (NPUs), field-programmable gate arrays (FPGAs), and application-specific integrated circuits (ASICs). Table 2.1 summarizes their main trade-offs in terms of programmability and portability, throughput and latency, power efficiency, and development cost/time to deploy. In brief, CPUs offer the highest flexibility and the most mature toolchains but are limited to deterministic line-rate processing at high speeds; NPUs provide protocol-aware acceleration with moderate programmability; FPGAs combine near-ASIC performance with reconfigurability at the cost of hardware design effort; and ASICs deliver the highest throughput and lowest latency with minimal programmability and long development cycles. The following subsections discuss each class in turn, highlighting their strengths and weaknesses in the context of deploying P4 programs.

Table 2.1 The difference between the P4 targets (inspired from [1])

Target	Throughput	Resources	Latency	Flexibility	Example
CPU	+	++++	$> 10\ \mu s$	++++	BMv2, T4P4S
NPU	++	+++	$5 \sim 10\ \mu s$	+++	Netronome SmartNIC
FPGA	+++	++	$< 2\ \mu s$	++	NetFPGA SUME
ASIC	++++	+	$< 2\ \mu s$	+	Intel Tofino

2.3.1 Software (CPU-based)

Software-based P4 targets execute packet-forwarding programs directly on general-purpose CPUs. The most widely used reference implementation is the Behavioral Model version 2 (BMv2), which serves as the de facto software switch for P4. In this model, P4 applications are compiled into an intermediate JSON representation that can be loaded into the switch at runtime. Developers can extend its functionality through external functions or by modifying its C++ source code. Although bmv2 is primarily intended for testing and functional validation rather than production deployment, it can still achieve non-trivial performance levels, reaching throughput rates of up to 1 Gbps in IPv4 LPM routing scenarios [8].

Another representative software target is T4P4S [13], which adopts a compiler-based approach. It translates P4 applications into target-independent C code that interacts with a network hardware abstraction library. A key design principle of T4P4S is the separation of hardware-dependent and hardware-independent functionality, which improves portability across platforms.

Overall, CPU-based P4 targets offer maximum programmability and ease of debugging, making them indispensable for prototyping, testing, and education. However, their limited throughput and high processing latency prevent them from serving as viable solutions for high-speed, production-grade networks.

2.3.2 ASIC

Application-Specific Integrated Circuits (ASICs) are highly optimized chips designed for high-performance packet processing, to implement only the essential operations required for this task. In practical deployments, network devices based on ASICs typically integrate a secondary general-purpose subsystem (e.g., CPU-based) to handle control, monitoring, and exceptional processing tasks that the ASIC cannot directly support. In this context, the fast path refers to processing executed by the ASIC, while the slow path is handled by the general-purpose subsystem.

A conventional ASIC is organized as a fixed pipeline consisting of sequential processing stages, with fast SRAM or TCAM banks used to store forwarding rules for the lookup operations in each stage. To overcome the rigidity of fixed-function ASICs, flexible and programmable architectures have been introduced, such as the Reconfigurable Match-Action Table (RMT) architecture [14] and the Protocol-Independent Switch Architecture (PISA) [15]. These designs enable network operators to reprogram low-level data plane behavior to support new protocols or custom applications entirely in hardware.

A prominent example is the Intel Tofino [16], which implements the PISA model and executes P4 programs at line rate regardless of program complexity. This is achieved through deep pipelining, where each pipeline can process hundreds of packets simultaneously, and parallelism across multiple pipelines. In addition to standard arithmetic and logical units, Tofino incorporates specialized hardware blocks, such as hash computation units, random number generators, and stateful objects (counters, meters, and registers), thereby expanding the range of programmable packet-processing capabilities [8, 9].

2.3.3 NPU

Network Processing Units (NPUs) are specialized processors designed for high-speed network packet processing tasks such as load balancing, encryption, and table lookups. Unlike general-purpose CPUs, NPUs are optimized for parallelism and typically consist of tens to hundreds of lightweight processing cores. While the computing power of each individual core is limited, the ability to execute many packet-processing tasks concurrently provides significant performance gains.

An example is the Netronome Network Function Processor (NFP), a programmable NIC featuring 72 cores (called micro-engines), each capable of supporting up to 8 concurrent threads. To reduce memory access latency, fast SRAM banks of a few hundred KBs are co-located with these micro-engines. The NFP architecture supports interface capacities of up to 2×40 Gbps Ethernet [9].

2.3.4 FPGAs

The combination of high performance and programmability makes FPGAs not only appealing for experimentation but also a viable alternative to expensive and restrictive ASIC designs in production environments. In networking, FPGAs are often integrated into network interface cards (NICs) to offload packet processing tasks from servers, thereby conserving valuable CPU cycles [9].

High-Level Synthesis (HLS) holds strong potential for enabling high-performance and flexible computing, particularly for packet processing, by bridging high-level programming (e.g., C++ and OpenCL) with spatial computing architectures such as FPGAs. Unlike traditional hardware description languages, HLS raises productivity while allowing hardware-specific optimizations essential for throughput- and latency-critical tasks. By exploiting spatial architectures, HLS enables application-specific dataflow that minimizes data movement and enhances energy efficiency. Its optimization opportunities include pipelining transformations

for predictable low-latency streaming, scalability transformations for exposing parallelism to handle high packet rates, and memory-enhancing transformations to maximize bandwidth utilization and optimize off-chip accesses. These techniques have shown dramatic speedups, sometimes several orders of magnitude, and allow developers to construct deeply optimized data paths tailored for packet streams. Together, these capabilities establish HLS as a powerful methodology for designing high-performance, energy-efficient, and adaptable packet processing pipelines on reconfigurable hardware [17, 18].

CHAPTER 3 LITERATURE REVIEW

3.1 Introduction

The growing demand for flexible and high-speed packet processing has positioned FPGAs as a promising platform for programmable data planes. To ground our work, we review prior research across four interconnected areas. We first examine FPGA-accelerated programmable data planes, which leverage reconfigurable parallelism to achieve the performance required by modern networks. One of the most critical parts of a packet processing engine is how to design and implement match-action tables, which require Content-Addressable Memories (CAMs), motivating our review of FPGA-based CAM architectures. At the programming level, we turn to the P4 conversion, now widely used to express data-plane behavior at a high level. Finally, we explore P4-to-FPGA transition frameworks, which bridge high-level programmability with efficient FPGA implementations, enabling the automated deployment of P4 applications on reconfigurable platforms. These works outline the state of the art and highlight open challenges in building efficient, FPGA-based programmable data planes.

3.2 FPGA-accelerated Programmable Data Plane

FPGAs have become an attractive target for programmable data planes due to their ability to provide energy-efficient and eco-friendly hardware solutions, which is increasingly important as sustainable networking gains attention. Unlike general-purpose processors, FPGAs can be optimized for specific network functions, making them highly suitable for demanding environments such as data centers and cloud platforms, where both performance and power efficiency are critical. Moreover, FPGAs enable the design of custom hardware tailored to particular use-cases like IoT, edge computing, and mobile networks. Such specialization allows them to deliver higher performance, lower latency, and improved flexibility compared to fixed-function ASICs or software-only solutions, thereby enhancing the overall efficiency and adaptability of modern programmable data planes [19].

The work in [20] introduces a P4-enabled packet-over-optical edge node capable of performing stateful traffic engineering and cybersecurity tasks directly in the data plane, thereby reducing reliance on the SDN controller and lowering latency. The system supports dynamic traffic offloading through P4 meters, optical bypass using P4 registers, and DDoS mitigation against TCP SYN floods by detecting and blocking attacks at the switch. The architecture was tested on BMV2 software switches and the NetFPGA-SUME board, achieving latency

as low as 5 μs, scalability up to 10,000 flows, and efficient hardware utilization, showing advantages over OpenFlow and ASIC-based SDN switches. Similarly, [21] proposed a P4-based BNG data plane for a CORD environment to meet the demands of large-scale telecommunications providers. They evaluated the design on programmable targets including NetFPGA, Netronome Agilio SmartNIC, and Intel Tofino, supporting up to 8192 users in practice. Results indicated that while Tofino efficiently managed 8192 users with only 16% SRAM and TCAM utilization, NetFPGA was limited to about 4096 subscribers. Their approach suggested scalability up to 35,000 subscribers with further P4 optimizations. As a continuation, [22] explored a hybrid platform combining Intel Tofino and NetFPGA with 10 Gbps, as well as an updated FPGA Ultrascale+ board with 100 Gbps bandwidth. Although the hybrid design utilized external DRAM for Active Queue Management, it was limited by DDR bandwidth, with practical throughput not exceeding 10 Gbps.

In [23], the authors introduce P8: P4 With Predictable Packet Processing Performance, a framework designed to estimate packet forwarding latency in P4 programs based on program complexity. Their approach carefully evaluates the role of P4 constructs, including parsing, header operations, and table lookups, to build a predictive latency model. This model is validated with sub-microsecond accuracy, enabling predictable design of high-performance P4 pipelines. The evaluation spans three distinct P4 targets, Netronome SmartNIC, NetFPGA-SUME, and the T4P4S DPDK software switch, using a 10 Gbps testbed with MoonGen for traffic generation and latency measurement. Experiments on over 75 P4 pipelines confirm the model's accuracy and generalizability, demonstrating that target-specific profiles can reliably predict latency across diverse devices.

For the work in [24], the focus is on accelerating frequency estimation sketches, such as the Count-Min sketch, through HLS-based optimizations on FPGAs. The authors design a Load-Store Queue (LSQ) architecture combined with pragma-level optimizations, which boosts throughput by more than 3× compared to baseline implementations. Additionally, a cache-based pre-processing step helps reduce dynamic power consumption. Evaluations on Intel Arria 10 and Stratix 10 FPGAs achieve update rates of 330M and 500M updates per second respectively, while consuming up to 80% of embedded RAM resources. This HLS-based methodology not only competes with but also matches or surpasses RTL-tuned designs, while offering ease of development at the C level and portability to structured ASICs or eFPGAs. The study in [25] continues this line of research with further HLS-based optimizations specifically tailored for Count-Min sketch implementations, enhanced by pragma-level fine-tuning. Unlike low-level RTL implementations, this approach remains accessible for developers at the C level, offering both ease of adoption and competitive performance that can be extended to other hardware platforms such as eFPGAs or structured ASICs.

In [26], the authors propose an architecture for memory-efficient mapping of P4 match/action tables to FPGAs using the DCFL (Deterministic Context-Free Language) algorithm. This method effectively balances processing performance with available memory resources, addressing one of the main challenges in implementing P4 match-action pipelines on hardware. Building on this line of research, [27] introduces a memory-optimized FPGA architecture for high-speed exact match packet classification. The key contributions include a unique parallel design capable of classifying multiple packets per cycle, efficient utilization of FPGA memory tiles without replication, and scalability to beyond 2 Tbps throughput while providing flexible trade-offs between throughput, memory, and logic consumption. The architecture achieves up to 99.7% of baseline throughput with only 25–40% of the memory footprint. Implemented in VHDL and evaluated on both Xilinx UltraScale+ and Intel Stratix 10 FPGAs, it leverages dual-port BlockRAM/M20K memories to sustain frequencies above 400 MHz. The system handles up to 40k IPv4 or IPv6 5-tuple rules with resource usage in the range of hundreds of BRAM/M20K tiles and approximately 70k–95k logic units. Its foundation lies in cuckoo hashing with parallel hash and distribution blocks, enabling real-world operation on 100 Gbps+ links.

Turning to protocol parsing, [28] presents the Dynamic Configurable Packet Parser (DCPP), an FPGA-based design aimed at low-latency, runtime-configurable protocol parsing in programmable networks. Unlike traditional fixed or multi-stage parser architectures, DCPP integrates protocol logic into a single state machine while introducing a parallel extraction-matching design and novel offset-container rules to improve field parsing efficiency. This approach reduces hardware costs and achieves throughput above 80 Gbps with only 36 ns of latency. The design is implemented on a Xilinx Zynq UltraScale+ FPGA (XCZU19EG) and validated using a Dell R740 server and IXIA traffic generator. Its architecture combines extract, offset, and key-generation units with a TCAM-SRAM module and a 1024-bit PHV, while runtime reconfiguration is supported via FlowMod messages for dynamic protocol updates.

Tiara [29] introduced a hardware acceleration architecture for stateful load balancing (LB) that is designed to be high-performance, scalable, and efficient. The platform integrates commodity servers, an FPGA, and a programmable switch, with the central idea of mapping different LB tasks to devices based on their capabilities. Specifically, Tiara divides the LB process into distinct tasks: packet encapsulation/decapsulation and other throughput-intensive tasks are handled by the programmable switch with its high packet processing throughput, while memory-intensive tasks that require large capacity and high bandwidth are mapped to FPGAs equipped with HBM. This architecture achieves significant performance, with results demonstrating 1.6 Tbps throughput, 80 million concurrent flows, 1.8 million new

connections per second, and latency below 4 microseconds in the fast path on a holistic server equipped with 8 FPGA cards.

In the work on eXtra Large Tables (XLT) [30], a hybrid platform combining a P4-capable ASIC and multiple FPGAs with DRAM was proposed to support large-scale tables for applications such as Carrier-grade Network Address Translation (CGNAT) and 5G User-Plane Gateways. The system stores several gigabytes of match-action tables in DRAM, accessible by the FPGAs. Incoming packets are first processed by the ASIC, which forwards them to the FPGA for lookups; the FPGA then returns the packets with the corresponding action information to the ASIC. This design achieves between 70 and 220 million lookups per second with table sizes ranging from 4 to 64 million entries. Despite its strong performance, the bandwidth is limited by the maximum throughput supported by the FPGAs. The use of a caching mechanism in the ASIC could mitigate this limitation, and such a strategy aligns with techniques for optimizing heterogeneous data planes. Another drawback is that the FPGA implementation relies on Verilog rather than P4, requiring manual updates whenever the P4 program changes—making the development process tedious and prone to errors.

Flightplan [31] addresses resource constraints by enabling the disaggregation of data plane programs across multiple hardware targets. When the resources of a single device are insufficient, Flightplan splits the program so that it can execute across additional targets. Beyond this, it can disaggregate programs across heterogeneous devices, such as ASICs and FPGAs, when a single class of hardware cannot provide the required capabilities. This approach increases performance, flexibility, and power efficiency by exploiting the complementary strengths of different platforms. Compared to conventional methods, Flightplan enhances P4 programs by dividing them into subprograms that execute across diverse data planes. However, the system still requires manual intervention in cases such as inserting segmentation annotations or handling syntax related to resource allocation, which introduces risks of human error and configuration delays.

A heterogeneous solution extending programmable switch ASICs with FPGAs was presented in [32], offering a pathway to accelerate a broader set of network applications. Their case study focuses on data deduplication, a technique that reduces storage overhead by eliminating duplicate data. A key component of this process is computing a hash, or "fingerprint," for each block of data in the stream. While fingerprints are typically calculated at storage nodes before data is written to storage devices, this work shifts the computation to accelerated modules (AM) within FPGAs. By offloading fingerprint generation to FPGA-based modules, the architecture enhances efficiency and scalability for storage-related applications.

In [33], an FPGA-accelerated framework is introduced for QoS-aware 5G network slicing

within the Edge-to-Core (E2C) segment. This framework runs on a P4 programmable data plane and combines hardware and software forwarding devices for efficient task management. By leveraging NetFPGA capabilities, it minimizes software reliance and enables the execution of network slicing with 16 distinct slices, supporting 512 users and 8,192 flows. Another related work, [34], addresses challenges in achieving ultra-reliable and low-latency communications in 5G networks, focusing specifically on the user plane function (UPF) through programmable data planes. Implementations on both software- and hardware-based platforms are reported, where the hardware prototype uses NetFPGA and achieves a latency of 6 μs, compared to 18 μs for the software-based instance running with DPDK [35].

Extensions to the P4 Portable Switch Architecture are presented in [36], where support for cryptographic hash functions was added and evaluated across three P4 target platforms: T4P4S (CPU), Netronome Agilio NFP-4000 Smart NIC (NPU), and NetFPGA SUME. In the FPGA case, the algorithms SHA3-512 and SipHash-2-4 were implemented, achieving maximum throughputs of 7.6 Gbps and 4.2 Gbps, respectively.

A different contribution is described in [37], which introduces an FPGA-based hardware-accelerated cryptographic solution leveraging P4. The implemented cryptographic functions include the symmetric cipher AES-GCM-256, the hash function SHA-3, and the digital signature scheme EdDSA. These functions reach throughputs of up to 4.51 Gbps, 24.26 Gbps, and 462 operations/s per component, respectively. The work relies on Netcope P4, which translates P4 into templated VHDL code and uses HLS to compile the actions and externs.

3.3 CAMs on FPGA

Research on FPGA-based CAM architectures has advanced significantly, with many works aiming to improve throughput, reduce power consumption, and enhance resource efficiency. One of the earlier efforts, the Hybrid Partitioned TCAM (HP-TCAM) [38], introduced partitioning techniques to control the exponential growth in resource demand caused by brute-force implementations. While this design improved memory efficiency, it required preprocessed ordered data and lacked the ability to support dynamic updates. Later, the Ultra-Efficient TCAM (UE-TCAM) [39] refined the approach by removing redundant memory structures, which reduced resource utilization. However, these designs still depended heavily on complex RTL development methodologies. An alternative strategy was proposed in [40], where Segmented Transposed Indicators RAM (STIRAM) was used for building deep and narrow BCAMs. This method stored segment indicators and patterns separately in RAM structures, enabling a two-cycle match operation with concurrent reading. More recently, [41] introduced FMU-BiCAM algorithms for FPGA-based binary CAMs, where CAM keys were

used directly as addresses, supporting two-cycle updates and reducing power by removing lookup tables and minimizing unnecessary SRAM updates.

Further developments explored LUTRAM-based CAM designs, including DURE-TCAM [42], D-TCAM [43], and ME-TCAM [44], which achieved faster updates and lower latency. The DURE-TCAM architecture employed FPGA LUTRAM primitives to enable simultaneous search and update operations, addressing the issue of blocking updates seen in SRAM-based CAMs. Similarly, the D-TCAM [43] approach emulated 6-bit TCAM entries with 6-input LUTRAMs, grouping 64 such blocks into a 48-byte D-CAM. These D-CAM blocks could be cascaded horizontally and vertically to increase width and depth, while redundant flip-flops were exploited as pipeline registers. In another effort, ME-TCAM [44] improved memory efficiency by using multipumping-enabled LUTRAM. It divided the TCAM table into Hybrid Partitions (HPs), each simulated with four LUTRAMs. Although memory efficiency improved with higher multipumping factors, throughput decreased as a trade-off. Building on these ideas, RPE-TCAM [45] introduced selective bank activation to lower power consumption by 40%, achieving one-cycle updates and supporting both BiCAM and TCAM modes. It also incorporated a backup CAM (BUC) to handle overflow conditions efficiently.

SRAM-based designs have also been explored for optimizing CAM architectures. The REST architecture [46] increased emulated TCAM capacity through virtual blocks (VBs), where SRAM was partitioned into multiple VBs to optimize address mapping. This approach offered a tunable balance between throughput and memory efficiency, depending on the number of VBs. Another solution, the multi-region SRAM-based TCAM proposed in [47], divided TCAM entries into index and data fields, mapping the index to SRAM addresses and storing data content separately, thereby reducing memory usage. In addition, [48] presented a power-saving FPGA-based TCAM architecture that used segmented match lines. Each line was divided into four nine-bit segments, activated sequentially to reduce dynamic power consumption without compromising functionality.

Flip-flop-based implementations offered another path toward efficient CAMs, with designs such as LH-CAM [49] and G-AETCAM [50] making use of FPGA slice registers to achieve very low latency and high-speed operations. These solutions were particularly effective in small-scale systems, though scalability was limited by routing complexity and hardware overhead. More recently, TeRa [51] proposed a ternary and range-based CAM optimized for packet classification. It introduced Ternary Match Logic (TML) with NAND-NOR encoding, a carry tree-based Range Match Logic (CTRC), and Match Inversion Logic to efficiently handle rules. TeRa was designed with two variants: TYPE1 tailored for ASICs, and TYPE2 adaptable for both ASICs and FPGAs, delivering improvements in priority selection and

memory utilization.

Flexible reconfiguration has also been integrated into FPGA-based CAM designs. A notable example is the reconfigurable match table (RMT) architecture presented in [52], which incorporated TCAM-based match tables for packet processing. Its three-layer structure allowed dynamic resizing and reconfiguration of match tables without modifying hardware. The design also featured a segment crossbar for packet match table (PMT) sharing and signal conversion, including MAND operations that combined match lines across TCAM PMTs.

Even with these diverse approaches, much of the research relies heavily on RTL-based methodologies. Although RTL provides detailed control over FPGA resources, it introduces complexity, demands significant development time, and lacks flexibility when adapting to evolving requirements. High-Level Synthesis (HLS), on the other hand, offers a more productive alternative by enabling faster development, improving portability, and simplifying design space exploration. While CAM designs have been widely investigated at RTL and lower abstraction levels, HLS-based CAM implementations remain underexplored, particularly in the context of match-action tables for packet processing engines.

3.4 P4 Conversion

The work in [53] introduces PSDN, a fine-grained pipeline parallelization compiler for P4 programs targeting FPGA-based network switches. Unlike conventional compilers that parallelize at the table level, PSDN decouples match and action functions, analyzes dependencies, and exploits fine-grained parallelism. Key techniques include dependency-based scheduling, stateless function prefetching, and function fusion to reduce synchronization overhead. On hardware, PSDN compiles P4 into PX programs and then into Verilog via the SDNet toolchain, evaluated on the NetFPGA-SUME board with a Xilinx Virtex-7 FPGA and four 10 Gbps ports. Results demonstrate 12.1% lower latency and 3.5% less resource use (LUTs, registers, BRAM) compared to prior methods.

Another perspective is provided by [54], which proposes HAL (Hardware-assisted Load Balancing), a system designed to improve energy efficiency and throughput in SmartNIC-equipped servers by enabling cooperative processing between the host CPU and the SNIC processor. HAL integrates a hardware-based load balancer (HLB) implemented on FPGA with an adaptive software policy to dynamically redirect traffic between processors. It achieves up to 31% higher energy efficiency and 10% higher throughput across network functions, while maintaining low latency. The system was evaluated on NVIDIA BlueField-2 (100Gbps, 8 Arm A72 cores) and Intel Xeon Gold 6140 CPUs, with extensions to CXL-SNICs for supporting

stateful functions. The HLB FPGA prototype consumed only about 1% LUT resources and less than 0.1W power, introducing negligible latency overhead. HAL reached 80Gbps throughput for functions like NAT and REM, with further potential for ASIC and eFPGA migration.

In [55], the authors present the Networking Template Library (ntl), a C++ class library for developing high-speed FPGA networking applications using Vivado HLS. Their methodology leverages modern C++ features, object-oriented design, and template-based components to enhance code reuse, modularity, and performance. Key abstractions include higher-order functions, specialized streams, header manipulators, data structures, and a scheduler, enabling reusable dataflow graphs for packet processing. The library is evaluated through a UDP stateless firewall and a key-value store cache, both achieving 40 Gbps line-rate with lower latency and reduced resource usage compared to existing solutions. While P4 provides simpler syntax for packet-processing tasks, ntl offers greater flexibility for complex algorithms, such as key-value stores, that are difficult to express in P4. Control-plane management is supported via an AXI4-Lite CPU interface.

NetCL [56] provides a unified programming framework for In-Network Computing (INC) that simplifies programming on programmable data planes. It introduces C/C++ extensions so developers can write kernel functions for processing in-flight messages, making the interface more similar to CUDA or OpenCL rather than low-level packet processing. NetCL hides networking details by offering APIs for communication and runtime mechanisms, allowing developers to focus on application logic. It supports essential features such as message-passing with compute semantics, explicit forwarding actions (e.g., drop, send, reflect), global and managed memory with atomic operations, lookup memory for match-action tables, and placement specifiers for multi-device programming. Compared to P4, NetCL addresses programmability challenges by removing the need for verbose, low-level code. A compiler translates NetCL kernels into P4, enabling execution on existing hardware like Intel Tofino. NetCL reduces code size by 8–12× compared to handwritten P4, while maintaining the same performance and resource efficiency on PDP devices, ensuring line-rate processing.

3.5 P4 to FPGA Transition

Research and industry efforts have introduced several frameworks to translate P4 programs into FPGA-compatible implementations, spanning both commercial solutions and academic proposals.

On the commercial side, tools such as AMD Xilinx VitisNetP4 [57] and Intel's P4 Suite [58] have significantly advanced P4 FPGA deployment. These frameworks enable network operators to define packet processing in high-level P4 code, which is automatically translated into HDL for FPGA implementation. Integrated with environments like Vivado and Quartus Prime, they streamline the software-to-hardware transition, greatly reducing design complexity. Despite their effectiveness for standard tasks, their proprietary nature restricts cross-platform portability, and they struggle with complex or stateful functions. Furthermore, integrating custom P4 externs often proves cumbersome, limiting design flexibility.

Academic approaches have also explored multiple pathways for mapping P4 to FPGAs. The P4-to-VHDL generator [59] produces high-speed packet parsers by translating P4 code into synthesizable VHDL. The work in [60] explores a templated HLS/C++ methodology that specifically targets the parser block, providing an alternative design path.

One of the promising initiatives was P4FPGA [61], an open-source extension of the P4 compiler that incorporates a custom backend for generating HDL code. Depending on application needs, P4FPGA supports various architectural options, leveraging the inherent flexibility and programmability of FPGAs compared to ASICs. It enables developers to design externs directly in HDL, but these hardware externs introduce high latency, creating challenges for efficient code generation. To address this limitation, P4FPGA implements an asynchronous processing model that allows additional packets to be processed in parallel.

Further exploration can be found in [62], which introduces a protocol-independent parser for SDN networks using a two-step P4-to-Python and Python-to-RTL conversion process. Another effort, the P4→NetFPGA workflow [63], simplifies FPGA programming for developers with limited hardware expertise. By leveraging the Xilinx P4-SDNet compiler and the NetFPGA SUME platform, it provides a rapid prototyping environment for packet processing applications. While effective, frameworks such as P4-to-VHDL [59] and P4FPGA [64] are now outdated or limited in scope, and platform-dependent solutions like P4→NetFPGA [63] reduce portability and flexibility.

A more refined toolchain is introduced in the P4→NetFPGA workflow [63], which provides a complete development environment for compiling and executing P4 programs on the NetFPGA SUME board equipped with four SFP+ ports [65]. This approach leverages the P4-SDNet compiler and the SDNet data plane builder from Xilinx, requiring a licensed Vivado suite. Notably, it supports custom externs implemented in HDL, as well as the integration of external IP cores as P4 externs. The framework supports P4 programs based on the SimpleSumeSwitch architecture, enabling functional extensions while maintaining FPGA performance.

Another development, P4-to-VHDL [66], introduces a converter that directly translates software-based P4 VNFs into equivalent FPGA circuits. This method allows user-defined VNF implementations to be virtualized and deployed on existing data center infrastructures, combining high bandwidth with hardware acceleration. Despite these advantages, the tool is closed-source and remains limited in applicability.

Further work in this domain is the High-Performance FPGA Framework (HPFF) [67], which introduces a Network Programmable Packet Processor (NPPP). This architecture unifies packet parsing, classification, and data processing within a single FPGA-based platform for SDN. By adopting P4 as the programming language, the framework eliminates the complexity of HDLs while enabling protocol-independent, flexible packet processing. The design incorporates a pre-processing stage to optimize parse graphs, traffic flow identification with P4 counters, and a hybrid control-flow model to minimize hardware use and maximize throughput. Implemented as a pipelined architecture on a Virtex-7 VC709 FPGA, HPFF operates at 320 MHz and achieves up to 2.9× higher speed compared to architectures like NetFPGA-10G and SUME. This demonstrates how combining P4 programmability with FPGA acceleration can yield a flexible, resource-efficient, and high-performance solution for packet processing.

Despite the progress made by both commercial and academic efforts, existing P4-to-FPGA solutions remain constrained by limited portability, reliance on proprietary toolchains or external IPs, partial support for modern P4 constructs, and insufficient integration of stateful data-plane functionalities. Moreover, many approaches either focus on specific pipeline components or impose fixed architectural assumptions that limit design flexibility and scalability. These limitations motivate the need for a fully HLS-based, modular, and extensible framework that enables transparent mapping of P4 semantics to FPGA pipelines while preserving performance, portability, and programmability. In response to these challenges, this thesis proposes HLSCAM, P4THLS, and an extended stateful data-plane architecture, which together address the identified gaps through templated design, automated workflows, and native support for both stateless and stateful packet processing on FPGAs.

CHAPTER 4 CONTENT ORGANIZATION AND RESEARCH APPROACH

4.1 Content Organization

This dissertation presents the cumulative outcomes of a structured research effort, where each major contribution was disseminated through a series of peer-reviewed publications. Each article corresponds to a distinct research phase and addresses one or more of the technical developments outlined in this work.

The first article [11] established the conceptual and experimental foundation of the research. It demonstrated the feasibility of implementing P4-based network applications directly on FPGA hardware through RTL generation. The study validated the basic translation of P4 constructs into hardware logic while exposing the main limitations of existing approaches, including limited portability, dependence on vendor-specific IPs, and the absence of automation. These findings motivated the transition towards an HLS methodology to enable automated, flexible, and portable FPGA-based data-plane design.

The second article [12] addressed one of the central contributions of this dissertation. It introduced the HLSCAM library, which enables the complete implementation of BCAMs and TCAMs in pure HLS. This work explored the design space of CAMs in terms of key width, table depth, resource utilization, and latency, providing a systematic evaluation of trade-offs between performance and area efficiency. The resulting library removed the reliance on proprietary RTL IPs, establishing a reusable and vendor-agnostic foundation for implementing lookup tables within FPGA packet processors.

The third article [4] corresponds to the second, third, and fourth contributions. It introduced the P4THLS framework, a templated and automated workflow that converts P4 programs into synthesizable HLS modules representing the parser, match-action tables, and deparser. This work also demonstrated the integration of the generated designs with the AMD Vitis design workflow and validated them on a high-end FPGA testbed. Through extensive experimentation, the framework was shown to achieve high throughput and consistent timing closure across multiple P4 applications, confirming the practicality of an HLS-driven design flow for real-world network systems.

The fourth article (submitted) represents the final stage of this research. It extended the P4THLS framework by introducing support for stateful objects and metadata propagation, enabling advanced network features such as flow metering, in-band telemetry, and queue

monitoring. This contribution advanced the framework toward self-contained and intelligent FPGA-based data planes, where packet analytics and control operations can be performed entirely in hardware without offloading tasks to the control plane.

4.2 Research Methodology

The research methodology adopted in this dissertation follows a systematic, iterative engineering process designed to bridge the conceptual and technological gap between P4's high-level programmability and FPGA's hardware realization capabilities. The primary objective was to design, implement, and validate a unified framework that could automatically translate P4 programs into efficient FPGA implementations while maintaining transparency, modularity, and scalability. The methodology was structured around four interdependent phases: i) Conceptual and analytical investigation; ii) Framework design and development; iii) Functional extension; and iv) Experimental validation.

4.2.1 Conceptual and Analytical Phase

The research was initiated with a detailed examination of programmable data-plane architectures to identify the structural and methodological gaps that prevent their efficient deployment on FPGA platforms. This investigation focused on understanding how P4 programs are parsed, compiled, and executed in hardware targets, and identified the bottlenecks preventing seamless deployment on FPGA platforms. The analysis revealed that existing P4-to-FPGA frameworks suffer from limited language support, dependency on vendor-specific IPs, and complex integration steps that hinder portability and reuse.

To translate these theoretical insights into practice, a proof-of-concept experiment was conducted to implement a P4-defined AGF using RTL synthesis. This design illustrated the feasibility of translating P4 programs into hardware logic but also highlighted several challenges, particularly in parser generation, table management, and inter-module synchronization. These findings motivated the pursuit of a higher-level synthesis methodology that would automate translation, preserve programmability, and simplify hardware reuse. This conceptual stage thus defined the research objectives and shaped the foundation for the development of the P4THLS framework.

4.2.2 Framework Design and Development

Following the initial conceptual analysis, the second phase focused on the systematic design and realization of a unified architecture for FPGA-based packet processing. This stage cul-

minated in the creation of the P4THLS framework, a modular, parameterized, and template-driven structure that represents the core elements of a programmable data plane. The architectural design process followed a hierarchical modularization strategy in which each component of the P4 data plane was implemented as an independent HLS module with configurable interfaces and parameters. This modularization enabled rapid customization, scalability, and clear separation between data processing and I/O management.

A significant technical challenge during this phase was the implementation of match-action tables, which require high-speed associative lookups typically handled by CAMs. Since HLS did not natively support CAMs, a dedicated HLS library, HLSCAM, was developed. HLSCAM provided synthesizable BCAM and TCAM modules using standard HLS constructs and enabled design-space exploration across dimensions such as memory depth, key width, latency, and throughput. The resulting design allowed fine-tuned optimization of resource usage while preserving high performance and portability across different FPGA platforms. This contribution laid the groundwork for subsequent integration within P4THLS, forming the lookup core of the framework's match-action engine.

4.2.3 Framework Extension and Functional Enrichment

After establishing the baseline architecture, the research expanded the P4THLS framework to include advanced features of the P4 language. This phase focused on integrating stateful objects (registers, counters, and meters) and metadata propagation, which are essential for enabling intelligent and autonomous network operations. The methodology involved reorganizing the pipeline structure to separate I/O interfaces from the core packet-processing logic. This structural adjustment eliminated synchronization stalls that can occur when external traffic interfaces interact with internal stateful elements, thereby improving determinism and throughput.

The integration of stateful objects enabled line-rate packet operations, such as counting, metering, and flow tracking, to be performed entirely in hardware. These functions were implemented in both direct and indirect configurations, allowing them to be linked dynamically to match-action tables or controlled via the external plane. Metadata propagation, on the other hand, enhanced visibility across processing stages by embedding per-packet contextual information such as timestamps, packet length, queue depth, and ingress/egress port identifiers. Together, these extensions provided a foundation for in-band telemetry and flow-aware control, representing a significant functional leap from the baseline design.

4.2.4 Experimental Validation

The final phase of the methodology was devoted to rigorous experimental validation of the proposed architecture. A dedicated FPGA testbed was developed to evaluate functionality, scalability, and performance under realistic network conditions. The experimental platform was based on an AMD Alveo U280 FPGA, interfaced with dual Intel XXV710 network adapters through high-speed transceivers. The TRex traffic generator was used to generate synthetic workloads and measure packet latency, throughput, and packet loss.

Each design iteration was synthesized using the AMD Vitis environment and tested across multiple P4 applications, including IPv4 forwarding, ACL filtering, and packet classification. Measurements were collected to assess logic utilization, memory usage, timing closure, and end-to-end throughput. Results confirmed that the P4THLS framework achieved deterministic high-speed operation with minimal resource overhead, sustaining multi-gigabit line rates across diverse workloads.

The validation process also provided iterative feedback that guided further template optimization, pipeline balancing, and synthesis pragma refinement. This experimental approach ensured that the proposed framework was not only theoretically robust but also practically deployable in real-world FPGA network acceleration environments.

CHAPTER 5 ARTICLE 1: NORMAL AND RESILIENT MODE FPGA-BASED ACCESS GATEWAY FUNCTION THROUGH P4-GENERATED RTL

M. Abbasmollaei, T. Ould-Bachir and Y. Savaria, (published in 2024-03-16). "Normal and Resilient Mode FPGA-based Access Gateway Function Through P4-generated RTL," 2024 20th International Conference on the Design of Reliable Communication Networks (DRCN), Montreal, QC, Canada, 2024, pp. 32-38, doi: 10.1109/DRCN60692.2024.10539150.

Abstract

Customizing packet processing is crucial in the evolving network landscape, especially with the rise of 5G telecommunications and beyond. Software-Defined Networking and programmable data planes, powered by the P4 language and FPGA-based platforms, offer dynamic network customization that can be used to implement resilient networks. With their high performance and programmability, FPGAs present cost-effective alternatives for diverse network applications, including offloading packet processing from servers. This paper introduces a configurable FPGA-based data plane implementing the Access Gateway Function (AGF). It offers a resilient operating mode to enhance network reliability and availability. The paper leverages the P4 language and the VitisNetP4 Intellectual Property to create RTL streams, enabling AGF on a pure FPGA target. The reported experimental results demonstrate that the proposed architecture can support 50K user flows with a resource utilization lower than 15% of that available in an Ultrascale+ FPGA (xcu280-fsvh2892-2l-e). This leaves massive logic resources available to incorporate fault mitigation techniques and spare streams needed to enhance resiliency. Moreover, the presented workflow maintains an average latency of approximately 9 microseconds for each downstream or upstream packet.

Keywords
Data plane programmability, P4 language, FPGA, RTL design, Resiliency, 5G, Access Gateway Function

5.1 Introduction

Quick adaptation is vital in the ever-evolving landscape of network technologies due to diverse applications, rising user demands, and emerging technologies like 5G telecommunications and edge computing. A significant novelty of 5G is the integration of fixed internet access

into the core of the 5G network, enabling fixed-mobile convergence (FMC) [68]. For this purpose, the Broadband Forum introduced the Access Gateway Function (AGF) to provide essential functions for interaction with the 5G core network while ensuring connectivity and management of access connections for user devices [69]. Software Defined Networking (SDN) and programmable data planes offer dynamic solutions, allowing organizations to optimize networks for various applications [70]. The P4 language [10] stands out as a powerful tool for defining data plane behavior in a concise and high-level manner.

Network devices encompass CPUs, FPGAs, and ASICs [2]. CPUs offer high flexibility and ample resources but have performance limitations. ASIC targets, such as Intel's Tofino switches, achieve high-performance packet processing through dedicated hardware. FPGAs are gaining recognition in data plane packet processing due to their adaptability, low latency, and ability to enhance network functions [71]. While ASIC switches provide high bandwidth, FPGAs offer a cost-effective alternative with competitive latency for various network applications. Integrating P4 with FPGA-based platforms presents an opportunity to develop a flexible and configurable RTL-based design capable of adapting to a wide array of traffic patterns and network conditions [66]. Moreover, they allow for implementing resilience techniques, such as deploying redundant processing streams or conducting statistical measurements.

This paper introduces an FPGA-based solution for deploying the AGF as an internal service in the data plane of the 5G core network by employing the P4 language as a source functional specification to obtain an RTL implementation. The proposed design is implemented using VitisNetP4, a tool recently introduced by AMD (Xilinx) [57], and formerly known as Xilinx SDNet. The proposed implementation offers a resilient operating mode that can be used to improve the reliability and availability of the network. The contributions of this paper can be summarized as follows:

- An FPGA-based solution that enables implementing the AGF as a P4 program executed on modern high-end Xilinx FPGAs is proposed;
- Two operating modes serving different purposes are introduced: the normal mode and the resilient mode;
- Key design parameters are evaluated for various numbers of supported user flows.

The remainder of this paper is structured as follows: Section 5.2 introduces AMD VitisNetP4 features and specifications, followed by related work. Section 5.3 then describes the proposed FPGA-based solution and its various operating modes. Next, Section 5.4 describes our

experimental setup, and the practical environment used to validate the proposed solution. It also compares and discusses our results. Lastly, Section 5.5 brings this work to a close by summarizing our main findings and highlighting promising areas for further research.

5.2 Background and Related Work

5.2.1 Access Gateway Function (AGF)

The development of 5G cellular networks aimed to surpass the data speeds of the 4G LTE networks while extending coverage, ensuring more reliable connections, minimizing latency, reducing energy consumption, and enhancing scalability. Several key entities collaborate within a 5G core network to enable the network's functionalities and services [70]. Figure 5.1 illustrates the positioning of each entity in a simplified manner. The 5G-RG module serves as the interface between user equipment (UE) and the 5G core network, while the AGF operates in the access network, bridging the gap between the radio access network and the 5G core network. The UPF is responsible for managing user data plane traffic within the 5G core network, while the AMF plays a critical role in access and mobility management, handling UE registration, authentication, and authorization procedures.

Makhroute *et al.* [3] introduce an implementation of the AGF developed using the P4 language and evaluate its performance when deployed on a Tofino switch. Their reported results demonstrate the impressive handling of a substantial number of sessions (approximately 400k) by a programmable P4 Tofino switch. Figure 5.2 provides a schematic representation of the AGF flow proposed in [3], featuring three distinct streams called downstream, upstream, and control streams, each comprising varying table sizes and widths. This paper extends the work by porting the code to an FPGA target, which is advantageous for remote deployments. Additionally, the proposed implementation incorporates a resiliency mode for enhanced robustness.

5.2.2 FPGA-based P4 5G Core Functions

The literature presents several FPGA-based P4 5G functions. In [33], an FPGA-accelerated framework for QoS-aware 5G network slicing within the Edge-to-Core (E2C) network segment is introduced. That framework offers an implementation running on a P4 programmable data plane. This architecture effectively combines hardware and software forwarding devices to manage tasks efficiently. Utilizing NetFPGA capabilities, that implementation reduces the reliance on software, enabling the system to execute network slicing with 16 distinct slices supporting 512 users and 8,192 flows. In another work, [34], challenges in providing ultra-

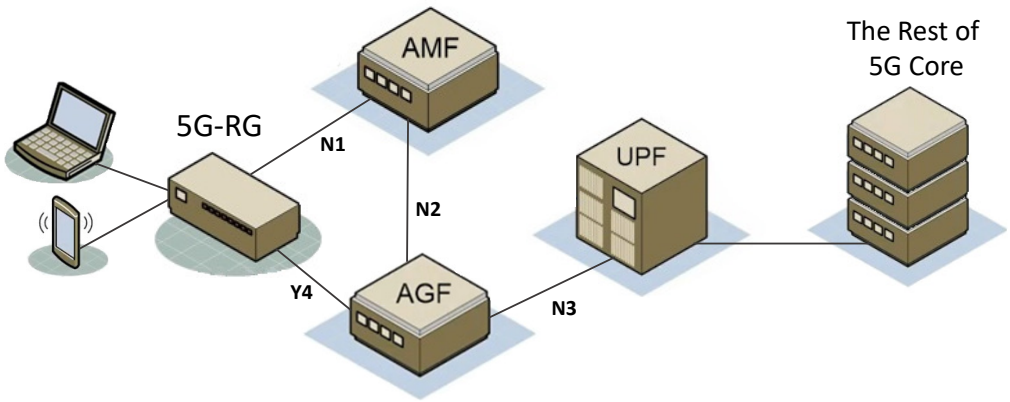

Figure 5.1 5G core network schematic, inspired from [2]

reliable and low-latency communications in 5G networks are discussed, focusing on the user plane function (UPF) using programmable data planes. Both software- and hardware-based platforms were implemented utilizing DPDK [35] and NetFPGA. The reported latency for the hardware-based prototype is 6 μs, compared to 18 μs for the software-based instance.

Previous studies have also explored Broadband Network Gateway (BNG). In [21], the authors introduced a P4-based solution implementing BNG on various programmable data planes. Their Tofino implementation supports up to 8,192 user flows with minimal SRAM and TCAM utilization, while the NetFPGA supports 4,096 subscribers. With P4 optimization, this technique can potentially expand to 35,000 flows. Following P4-BNG, OpenBNG [22] employed an Ultrascale+ FPGA instead of NetFPGA, alongside a hybrid architecture incorporating an Intel Tofino switch and an FPGA.

5.2.3 P4-based HDL

Several alternatives exist to convert P4 programs into synthesizable Hardware Description Language (HDL) code. For example, P4FPGA [61] converts P4 programs to Bluespec programs, which are then used to generate Verilog source code. P4-TO-VHDL [59] directly generates VHDL code for specific FPGA architectures, although it currently only supports $P4_{14}$. More recently, P4 to FPGA [66] is a framework based on templated libraries proposed for converting P4 programs to VHDL.

Commercial alternatives are also available. VitisNetP4 IP [57] is an updated version of Xilinx SDNet. It offers several options to adjust the IP to fit various performance requirements. With tools of this type, data bus width and clock frequency are key parameters determining the resulting implementations' throughput. In addition to these parameters, the resource

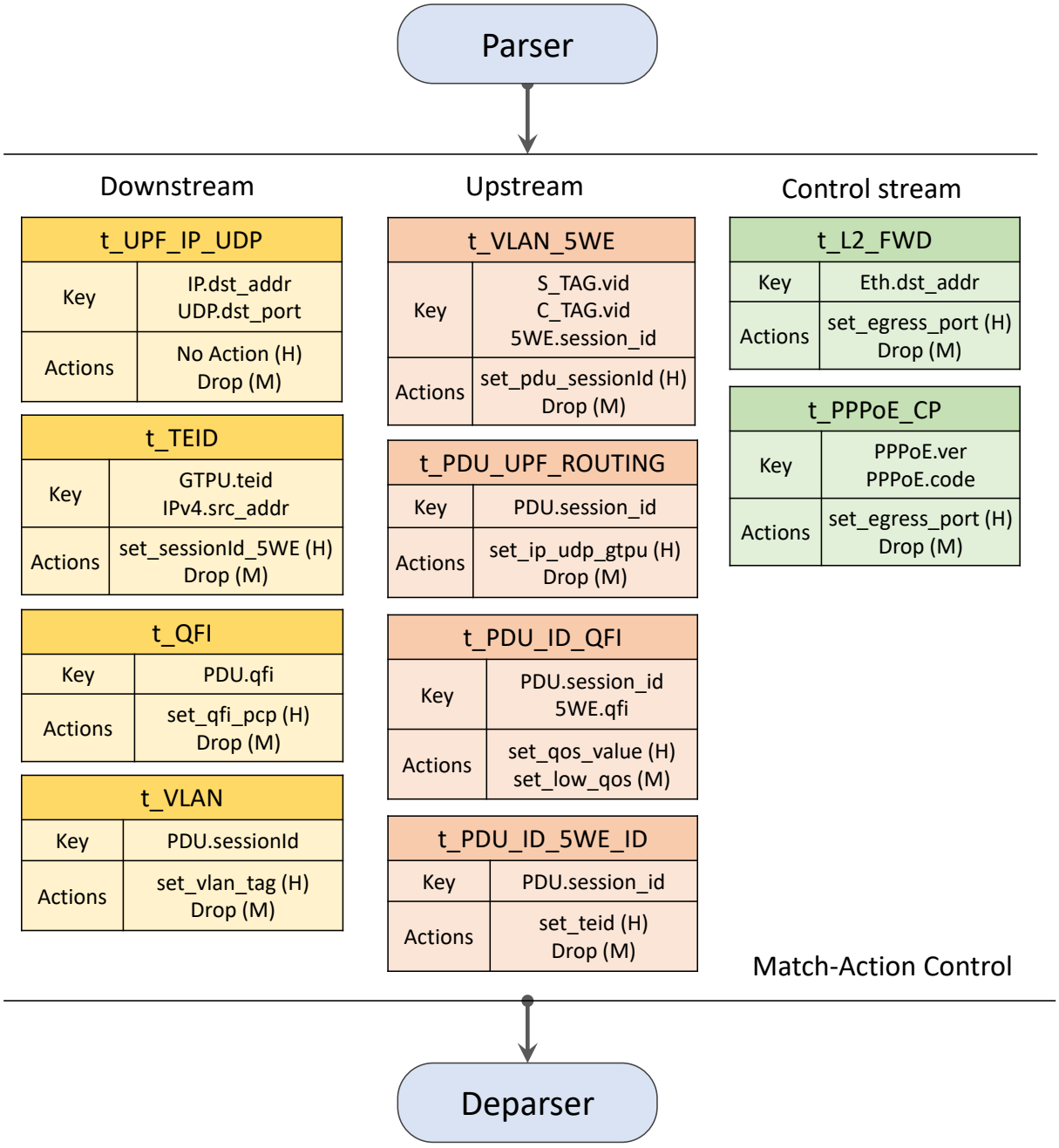

Figure 5.2 The data flow of the AGF application, adopted from [3].

type used for tables and other miscellaneous options can affect the latency.

Given that each P4 program has different internal states for parsers, match-action, and deparser, the latency may vary depending on incoming packets. However, the tool considers the longest path and reports a fixed value as the module's latency. The value is valid until there is no back pressure. Back pressure is the accumulation of data in a switch's pipeline

when the buffers are full and unable to receive additional data [72].

5.3 Proposed Architecture

5.3.1 FPGA-based Architecture for the AGF

The AGF is designed to accommodate diverse protocols and connects the Edge to the Core. Fig. 5.3 illustrates an abstract view of the data plane architecture and the positioning of the proposed hardware in 5G orchestration. As discussed earlier, the AGF is divided into three parts: one for processing control plane packets, a second one for downstream (DS) packets, and a third one for upstream (US) packets. Control plane packets travel through the PCIe interface. P4 pipelines are generated for the US and DS, respectively. The proposed architecture considers primary and standby pipelines, as shown in Fig. 5.3.

For this work, we assumed that UPF and AMF are located at the 5G core, and the designed AGF handles incoming and outgoing packets while performing the defined operations on them. Although the control plane traffic constitutes a minor portion, it holds critical significance during authentication and session establishment. The protocol stack for this traffic type encompasses Ethernet, double VLAN, PPPoE, and PPP [3]. The primary role of the AGF in controlling plane traffic is to detect incoming and outgoing streams and route them to the AMF unit.

While the proposed implementation enables parsing, matching, and deparsing of this type of packet, it is limited by the number of physical port interfaces; therefore, control plane packet routing remains a responsibility of the 5G core units. In contrast, user data is carried through data plane packets, which constitute the majority of the traffic and are primarily transmitted through interfaces V and N3. The protocol stack for this traffic type varies depending on the interface. Traffic originating from the V interface encompasses Ethernet, double VLAN, and 5WE. Conversely, traffic directed towards the N3 interface involves a protocol stack including Ethernet, IP, and GTP-u.

5.3.2 Implementing P4-based Pipelines on FPGA

The VitisNetP4 tool can take P4 programs and generate HDL modules in an encrypted format. Specifically, the tool creates Xilinx Pipelines for each P4 code, constituting dedicated packet processing units with three main components: Parser, Match/Action, and Deparser. While it simplifies the deployment of P4 programs, it imposes constraints on certain built-in P4 language capabilities, such as division and modulo operators, specific data types, and the

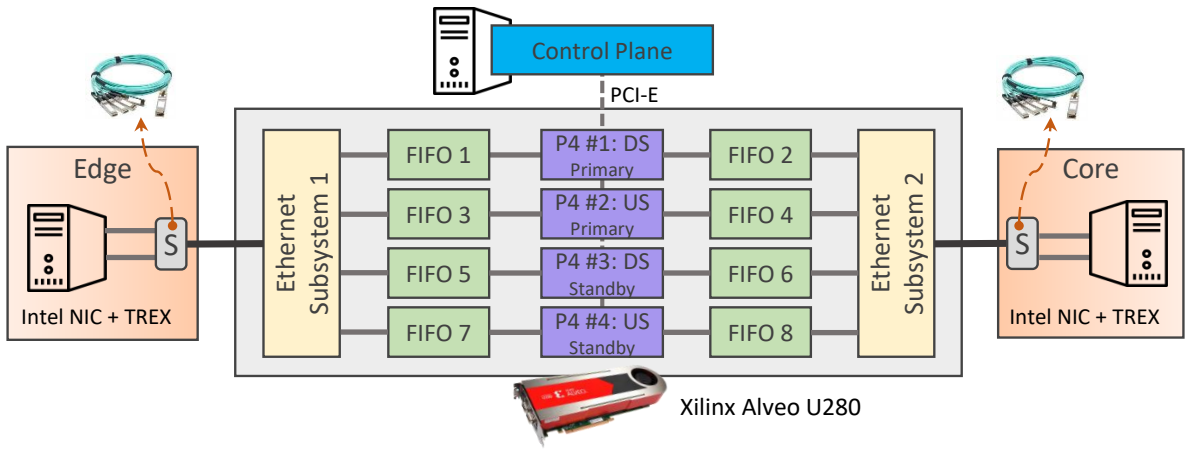

Figure 5.3 Experimental setup of the proposed FPGA implemented AGF architecture

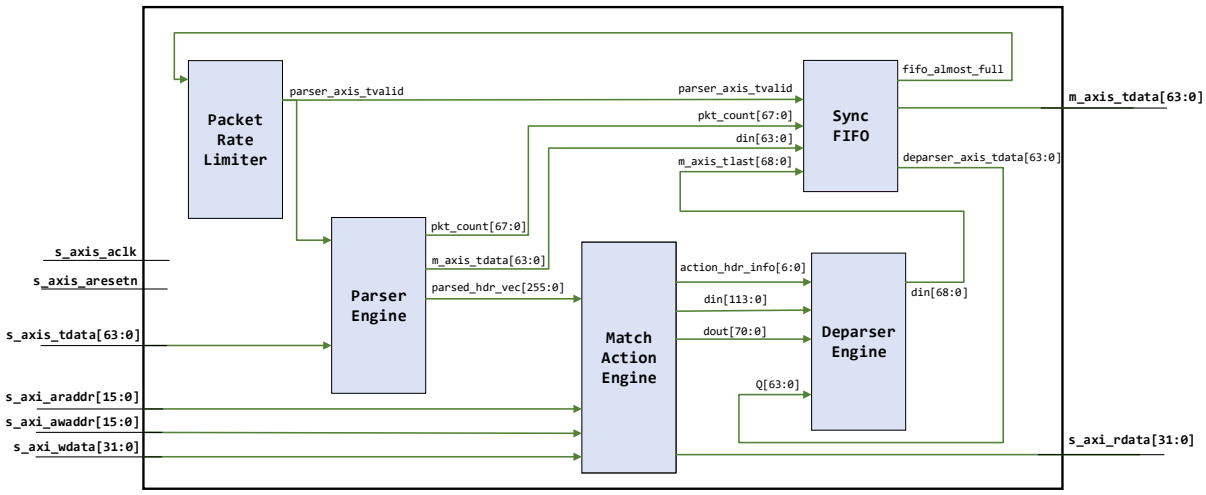

Figure 5.4 The synthesized schematic of a P4 IP instance within the AGF implementation

parser's 'lookahead' feature.

As a result, the initial task is to ensure compliance with these constraints in the generated P4 code. Subsequently, all modules are encapsulated within a kernel entity to enable integration into the Vitis Development Workflow, a contemporary approach for flexible implementation and configuration of hardware-accelerated designs on the latest FPGAs. Regarding connectivity with other IPs and components, such as Ethernet subsystems, the VitisNetP4 IP offers an AXI-4 stream, a standard bus in Xilinx FPGAs. Additionally, it provides an AXI-Lite bus for connecting to the control plane through a PCIe interface, facilitating the management of tables, counters, and other configurable elements.

Fig. 5.4 presents the synthesized schematic of the AGF P4 IP, including Parser, Match/Action, Deparser engines, Packet Rate Limiter, and Sync FIFO. Non-essential wires have been removed from the schematic to make the crucial nets and buses easier to see. The Packet Rate Limiter block gets feedback from the Sync FIFO block called `fifo_almost_full`, which notifies the input stream to slow down or stop receiving data chunks. The Packet Rate Limiter also emits a signal called `parser_axis_tvalid` to inform relevant components handling the headers. The Sync FIFO block, placed at the end of the pipeline, is responsible for syncing the packet segments and streaming out through a master AXI-4 stream. Furthermore, the AXI-Lite stream signals beginning with `s_axi_` are associated with the control plane access to the IP for retrieving or manipulating tables and memory elements.

5.3.3 Resilient Operating Mode

Numerous threats pose risks to the security and reliability of 5G networks. In this discussion, we explore the enhancement of the resilience of the proposed FPGA-based Access Gateway Function (AGF) protocol within 5G networks. Firstly, general threats and vulnerabilities to the network infrastructure and potential countermeasures to mitigate these risks are discussed. Subsequently, we narrow our focus to a specific threat posed by Distributed Denial of Service (DDoS) attacks, which can overwhelm incoming links, and propose a potential solution to address this particular threat.

General Threats and Solutions

We will first provide an overview of the vulnerabilities associated with 5G networks, emphasizing the critical role of AGF service and its susceptibility to cyber threats. Next, we will denote the potential countermeasures that could be implemented besides the proposed architecture to build a reliable FPGA-based environment to deploy AGF collaborating with the 5G network.

One common attack method involves packet flooding, where a malicious source floods a specific server or service with an overwhelming traffic volume, hindering its ability to effectively process genuine service requests. This type of attack could saturate the AGF streams with an excessive number of packets, resulting in network congestion and a decline in service quality. A possible solution involves detecting abnormal behavior and flooding traffic by adding monitoring elements to the data and control planes. P4 provides tools to track packets with specific identifiers using built-in counters. The control plane can periodically collect these metrics, analyze traffic patterns, and take action by updating data plane tables to identify and drop suspicious packets. This approach enables the system to analyze traffic using static

algorithms or AI-based methods to improve detection accuracy. While the ideal approach is to detect suspicious traffic within the data plane, this is often impractical due to processing limitations and resource constraints. To address this, the AGF architecture includes standby lanes that can be activated when the primary lanes are overwhelmed with abnormal traffic. In such cases, congestion detectors and multiplexers can be placed before each port to divert traffic to standby lanes until the primary pipelines become available again.

A more specific threat to AGF services involves attackers injecting manipulated or forged packets to establish many user flows and allocate multiple session IDs. If the number of allocated flows surpasses the capacity of the associated tables within the implemented AGF, legitimate users may be prevented from being handled, delays may occur, or service may be denied to those genuinely seeking access. To address this threat, one effective approach could involve identifying flows with minimal utilization. The frequency with which each flow occurs could be measured using a probabilistic-based method like the Count-Min sketch, flagging less frequently used flows as potential attempts to saturate the tables used in the P4-based AGF. The data plane could measure usage frequency, and the control plane would retrieve statistics to update VitisNetP4 IP tables and remove infrequent entries. Coordination with other services and servers in the 5G core networks is essential to prevent duplication and creation of these entries multiple times.

Specific Threat and Solution

In addition to the standard functionality of the FPGA-based AGF within the 5G core network, a resilient mode could be integrated to enhance availability and mitigate the impact of Distributed Denial of Service (DDoS) attacks. A light version of the DDoS attack detection and mitigation approach inspired by [73] can locally create a protection layer to prevent volumetric types of attacks. This resiliency approach operates by monitoring incoming packets and identifying their respective flows. This process involves leveraging the existing tables within the AGF, coupled with a set of Extended Finite State Machines (EFSMs) responsible for tracking the status of each flow. By effectively categorizing and counting incoming packets per flow, we can swiftly detect anomalies indicative of DDoS attacks when the packet rate of each flow exceeds a predefined threshold. Several mitigation strategies can be employed to safeguard the network's integrity and availability upon detecting suspicious patterns. These strategies may include throttling or dropping packets from identified malicious sources, dynamically reallocating resources to prioritize legitimate traffic, or triggering alerts for further analysis and response by network administrators. In our case, it drops packets of suspected flows until each flow returns to its normal state.

When identifying a sequence of packets as an attack, timing and quantity constraints are the two measures to consider. Timing restrictions include an interval threshold applied to each flow's subsequent packets. On the other side, quantity constraints relate to a cutoff point that restricts the number of successive packets that arrive for every flow. The state diagram for attack detection and mitigation, considering temporal and quantity constraints, is shown in Figure 5.5. The letters "T" and "C" stand for triggering the time and amount, whereas the letter "n" before them denotes not triggering them. The initial state, `SAFE`, is one in which no restrictions are imposed by traffic flow. The flow enters the `SUSPECT` state when the time interval between successive packets from each flow surpasses the timing threshold. Counting the consecutive packets with a short time interval falls to a counter. It stays at `SUSPECT` until both restrictions are released, which takes a predetermined amount of time, at which point it returns to its initial state. However, if the quantity threshold is exceeded, the status changes to `ACTIVE`, and the packets will be dropped. The diagram changes to `COOLDOWN` if the current state of `ACTIVE` persists without any triggers, then it reverts to its initial state after a predetermined time.

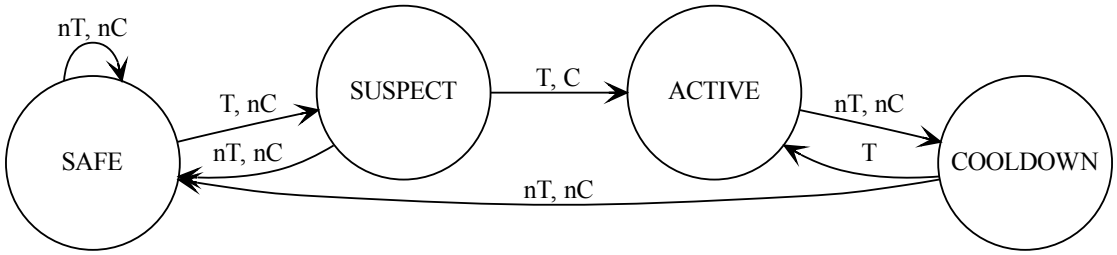

Figure 5.5 The EFSM diagram for detecting and mitigating volumetric DDoS attacks

A P4 code snippet showing how the detection and mitigation mechanism works in the data plane is given in 5.1. The table `t_Limiter` is added to update the status of each flow by matching `meta.flow_id`, which is set in the last match-action for each accepted packet when all the necessary conditions are met. To avoid showing unnecessary details, these conditions are compacted in a single variable called `condition` used in line 37. The control block measures the time interval between the current and the last packet of the same flow, thanks to the timestamp field in the standard metadata of the VitisNetP4 IP (lines 41 to 43). Then, it reads the dynamic threshold (line 46) that could be determined by the control plane and checks whether the interval is less than the threshold or not (line 47). If so, the corresponding counter counts up. The control plane is responsible for pooling the counters and updating the status of each flow in the table `t_Limiter`.

Listing 5.1 The extended P4 code enabling the proposed EFSM in the data plane

```
// <Include architecture headers here>
typedef bit<16> Index_t;
const bit<32> NUM_FLOWS = 50000;
const bit<2> STATE_SAFE = 0;
const bit<2> STATE_SUSPECT = 1;
const bit<2> STATE_ACTIVE = 2;
const bit<2> STATE_COOLDOWN = 3;

// <Define headers, metadata, parser here>

control AGF_Processing (
                inout headers hdr,
                inout metadata_t meta,
                inout standard_metadata_t smeta) {
  // Define tables and actions here

  action apply_efsm (bit<2> status) {
    if      (STATE_SAFE)     { /* Do nothing */ }
    else if (STATE_SUSPECT)  { /* Do nothing */ }
    else if (STATE_ACTIVE)   { drop(); }
    else if (STATE_COOLDOWN) { drop(); }
    else                     { drop(); }
  }

  table t_Limiter {
    key = { meta.flow_id : exact; }
    actions = { apply_efsm; NoAction; }
    size = NUM_FLOWS;
    default_action = NoAction;
  }
  Counter<bit<8>, Index_t>(NUM_FLOWS, CounterType_t.PACKETS)
     FlowPktCounter;
  register<bit<64>>(NUM_FLOWS) FlowTimeTag;
  register<bit<16>>(1) thInterval;

  apply {
    if (condition) {
      // set meta.flow_id by a match-action based on the session id
      ...
      t_Limiter.apply();
      Index_t idx = meta.flow_id;
      bit<64> lastTime;
```

```
42        FlowTimeTag.read(idx, lastTime);
43        bit<64> curTime = smeta.ingress_timestamp;
44        FlowTimeTag.write(idx, curTime);
45        bit<16> th;
46        thInterval.read(1, th);
47        if (curTime - lastTime < th)
48          FlowPktCounter.count(idx);
49      }
50    }
51  }
52
53  // <Define deparser and pipeline here>
```

While identifying various DDoS attacks demands thorough and intelligent traffic analysis, relying solely on a fixed threshold for packet count per flow to restrict traffic may not address all DDoS attacks. However, this approach presents a promising avenue for developing more sophisticated mitigation techniques to enhance the resilience of FPGA-based data planes.

We showed that the proposed FPGA-based workflow for implementing the AGF protocol has the potential to be integrated with this resilient mechanism to make it more robust. Still, implementing such a solution remains in future work due to VitisNetP4's limitations, like not supporting appropriate stateful objects. These limitations can be resolved with user externs.

5.4 Evaluation

5.4.1 Experimental Setup

A Xilinx Alveo U280 board, also known as NetFPGA PLUS, which includes an Ultrascale+ FPGA SoC and two QSFP28 ports with a maximum throughput of 100 Gbps, is used for the AGF deployment on the data plane. DPDK [35] and TReX [74] environments are utilized to produce appropriate traffic, and packets are injected via the network using two Intel network interface cards (NICs) with two 10 Gbps ports. As shown in Fig.5.3, the FPGA has two QSFP28 physical interfaces, each supporting 100 Gbps with four 25 Gbps lanes. One interface is dedicated to the edge side, while the other is to the core side. Four lanes of each QSFP link are split using a splitter cable to allow us to inject and capture traffic using SFP-supported NICs.

5.4.2 Implementation Results

Table 5.1 indicates the resource utilization of the FPGA for the entire platform, including VitisNetP4 IPs, Ethernet subsystems, FIFOs, and other necessary peripherals. The table reports the results for three different table configurations. The implemented design could support various distinct user flows according to the AGF table sizes, and the ability to scale the number of supported subscribers is a key point in 5G networks. Hence, the proposed platform was evaluated when supporting 10k, 30k, and 50k user flows. To do so, we set the "Auto" option in VitisNetP4 IP configuration for selecting memory resources and 64-bit for AXI4 streams in synthesis. The utilized FPGA resources are almost identical, except for UltraRAM (URAM) units. As anticipated, the utilization of URAM grows with the expansion of table sizes. It should be noted that this increase arises from the varying size and entry width of each table in the P4 code, causing each table to only use part of the allocated URAM elements. Consequently, certain portions of URAMs often remain wasted.

The overall resource utilization is below 15%, indicating that expanding the AGF application to accommodate more subscribers largely relies on high-capacity on-chip memory resources, such as URAMs. Utilizing off-chip memory resources like DDR and HBM presents an opportunity to enhance the capabilities of the FPGA-based system. It is worth noting that using external memory sources can introduce additional latency to the entire platform, and this aspect will be explored further in our future work.

At the time of completing the paper, no implementation results for the resilient mechanism were available due to the absence of support for stateful objects in the Vitis Networking P4 Software. However, this issue can be addressed using user externs and is designated as a

Table 5.1 FPGA resource utilization for different numbers of supported user flows

Supported User Flows	LUT (1,303,680)	LUTRAM (600,960)	FF (2,607,360)	BRAM (2016)	URAM (960)
10,000	179,121 (13.74%)	14,425 (2.40%)	236,345 (9.06%)	241.5 (11.98%)	32 (3.33%)
30,000	180,713 (13.86%)	15,026 (2.50%)	238,002 (9.13%)	241.5 (11.98%)	64 (6.67%)
50,000	180,749 (13.86%)	14,978 (2.49%)	238,081 (9.13%)	241.5 (11.98%)	128 (13.33%)

Table 5.2 The comparison of latency and the number of user flows that each work supports

Work	5G Service	# of user flows	Device	Overall latency
P4-BNG [21]	BNG	8,192 (max: 35,000)	ASIC, SmartNIC, FPGA	1 μs, 22 μs, 5.08 μs
OpenBNG [22]	BNG	8,192 (max: 35,000)	ASIC, SmartNIC, FPGA, Hybrid	1.17 μs, 22 μs, 5.08 μs, 7.39 μs
[3]	AGF	286,500 (max: 400,300)	ASIC	Not reported
This work	AGF	50,000	FPGA	9 μs

future endeavor.

5.4.3 Scalability Results

Table 5.2 presents a comparative analysis of the proposed data plane with three recent works. The first two projects implemented BNG services on various devices, while the third implemented AGF on Intel Tofino. P4-BNG [21] and OpenBNG [22] support up to 8,192 subscribers and suggest that 35,000 sessions are sufficient for each 5G central office. In contrast, the work in [3] demonstrates support for 286,500 and 400,300 distinct user flows in normal and optimized versions, respectively. It consumes nearly the entire SRAM resource across all pipeline stages of a Tofino, and it fully utilizes the TCAM resources in some pipeline stages, limiting the scalability of the data plane. Conversely, our proposed pure FPGA design uses less than 15% of resources and can support 10,000 to 50,000 user flows, maintaining an average latency of approximately 9 microseconds for each downstream or upstream packet.

Supporting 50K user flows should be sufficient for a small-scale AGF solution comparable to the other two previously reported BNG works that only support 8192 flows with the ability to extend to 35K flows. Our work supports fewer user flows compared to the Tofino-based Makhroute's work. However, it offers the ability to add redundant streams and additional logic to handle congestion and allows increasing the user flows at the cost of using more URAM resources, which provides for a highly flexible solution considering that the utilization of URAM is below 13.3% on the target platform, The proposed FPGA solution can be scaled to adjust the FPGA size and resource requirements according to the application requirements, which is not possible with a complex ASIC such as the Tofino.

The measured latency represents the time a packet takes to traverse from the edge to the core.

Interestingly, latency remains relatively consistent across different table sizes, ranging from 10,000 to 50,000 users, with only a negligible increase in the number of clock cycles that is barely perceptible. While our approach exhibits slightly higher latency than the pure FPGA designs in P4-BNG and OpenBNG, it offers superior scalability with significantly more available resources. The primary factors contributing to latency are cable length and queuing delays, influenced by the quality of experimental materials, including splitters, sockets, and PCs. Despite these factors, the processing latency reported by the tool remains below 600 nanoseconds. Furthermore, our proposed architecture introduces a resilience mechanism, enabling the data plane to operate in a more reliable environment by utilizing standby streams.

5.5 Conclusion

This research proposes a pure FPGA-based solution capable of generating RTL streams based on the AGF. We used P4, as a high-level programming language for customizing packet processing on the data plane, and the associated IP in the FPGA design environment to create an RTL implementation of the AGF as a network function in 5G core facilitating the connectivity and management of access connections for user devices. Moreover, the paper considered the ability to increase the resiliency of packet processing by placing spare streams in standby mode. The spare streams could start operating in the event of overwhelming the primary streams. Our evaluations indicate that the proposed platform can support 50K user flows while occupying less than 15% of the FPGA resources. Future work will examine integrating the proposed design with advanced methods to improve resiliency.

CHAPTER 6 ARTICLE 2: HLSCAM: FINE-TUNED HLS-BASED CONTENT ADDRESSABLE MEMORY IMPLEMENTATION FOR PACKET PROCESSING ON FPGA

M. Abbasmollaei, T. Ould-Bachir and Y. Savaria, (published in 2025-04-23). "HLSCAM: Fine-Tuned HLS-Based Content Addressable Memory Implementation for Packet Processing on FPGA," Electronics, 14(9), 1765, doi: 10.3390/electronics14091765.

Abstract

Content Addressable Memories (CAMs) are pivotal in high-speed packet processing systems, enabling rapid data lookup operations essential for applications such as routing, switching, and network security. While traditional Register-Transfer Level (RTL) methodologies have been extensively used to implement CAM architectures on Field-Programmable Gate Arrays (FPGAs), they often involve complex, time-consuming design processes with limited flexibility. In this paper, we propose a novel templated High-Level Synthesis (HLS)-based approach for the design and implementation of CAM architectures such as Binary CAMs (BCAMs) and Ternary CAMs (TCAMs) optimized for data plane packet processing. Our HLS-based methodology leverages the parallel processing capabilities of FPGAs through employing various design parameters and optimization directives while significantly reducing development time and enhancing design portability. This paper also presents architectural design and optimization strategies to offer a fine-tuned CAM solution for networking-related arbitrary use cases. Experimental results demonstrate that HLSCAM achieves a high throughput, reaching up to 31.18 Gbps, 9.04 Gbps, and 33.04 Gbps in the 256×128, 512×36, and 1024×150 CAM sizes, making it a competitive solution for high-speed packet processing on FPGAs.

Keywords
Content-Addressable Memory; FPGA; High-Level Synthesis; packet processing

6.1 Introduction

Content-Addressable Memories (CAMs) are critical components in high-performance computing and data analysis. They enable fast data lookup operations, which are essential for many applications. For example, CAMs are used in packet routing and switching [51, 52]. They also play a role in DNA sequence analysis [75, 76]. In addition, CAMs support machine learning and data analysis, especially with approximate CAM designs [77].

Unlike traditional Random-Access Memories (RAMs), which access data using specific addresses, CAMs enable parallel content search. This significantly speeds up lookup operations and is especially crucial in data plane packet processing. Tasks like IP address lookup, access control, and flow classification demand fast and efficient search mechanisms to meet strict throughput and latency requirements [78].

Various CAM types serve distinct roles within packet processing pipelines. Binary CAMs (BCAMs) are optimized for exact match lookups, making them ideal for applications such as MAC address tables, as illustrated in Figure 6.1. Ternary CAMs (TCAMs) support wildcard-based matching, allowing flexible rule definitions that are indispensable in applications such as access control lists (ACLs) and packet classification. Additionally, a specific type of TCAM could only consider prefix matching to handle Longest Prefix Match (LPM) operations, which are crucial in IP routing. In LPM, each subnet in a network defines a range of IP addresses with a common prefix, and STCAMs efficiently match these prefixes to determine the most specific routing entry for a given IP address. This specialized functionality makes STCAMs highly effective in network IP matching and routing table lookups, where hierarchical addressing schemes are prevalent.

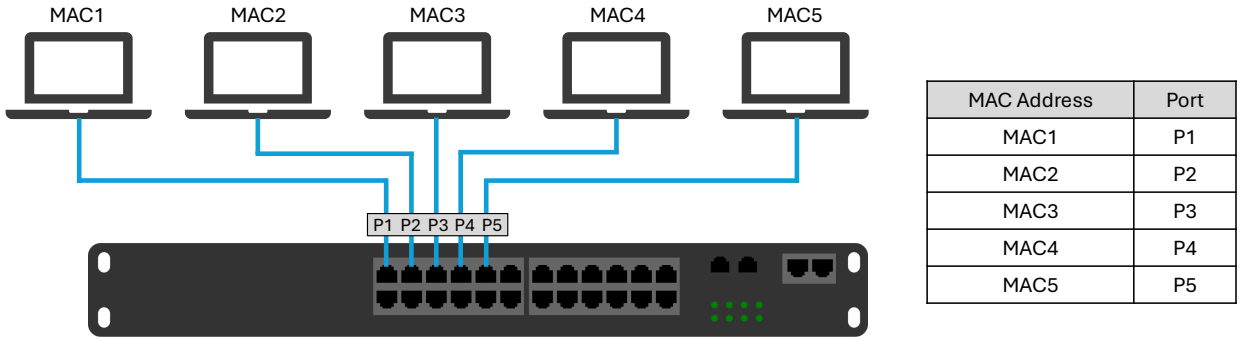

Figure 6.1 The use of CAM tables in L2 switching.

Despite their advantages, implementing CAMs for high-throughput environments presents several challenges. Traditional CAM designs, often realized through Application-Specific Integrated Circuits (ASICs), offer excellent performance but suffer from high development costs and lack flexibility for evolving network requirements. The adoption of Field-Programmable Gate Arrays (FPGAs) for CAM implementations has surged due to their inherent parallel processing capabilities and reconfigurability, offering flexibility over traditional Application-Specific Integrated Circuits (ASICs). FPGA-based CAMs are typically realized using four primary on-chip memory types: Ultra RAM (URAM), Block RAM (BRAM), Lookup Table RAM (LUTRAM), and flip-flops (FFs). Each memory type presents trade-offs in terms

of resource utilization, power efficiency, and scalability. For example, BRAM-based CAMs offer high density but suffer from longer update latencies, whereas LUTRAM-based CAMs provide faster updates at the cost of increased routing complexity. FF-based CAMs, though less scalable, excel in applications requiring minimal hardware overhead and rapid updates.

FPGAs offer a reconfigurable alternative that can provide CAM performance while providing additional adaptability. However, traditional Register-Transfer Level (RTL) design methods for implementing CAMs on FPGAs present several challenges. These approaches are complex, time-consuming, and error-prone, which hinders rapid prototyping and design space exploration. High-Level Synthesis (HLS) addresses these limitations by introducing a new design paradigm. HLS allows hardware description using high-level programming languages like C/C++, significantly simplifying the development process.

HLS also facilitates accelerated development cycles, improved design portability, and simplified integration with software-driven workflows. While HLS has demonstrated its potential in various application domains, to the best of our knowledge, no prior work specifically addresses the design and implementation of CAM architectures optimized for packet processing using HLS.

This gap in the literature motivates the present paper, which proposes HLSCAM (`https://github.com/abbasmollaei/HLSCAM`), an HLS-based approach to implement CAM architectures optimized for data plane packet processing on FPGAs. The contributions of the paper include the following:

- It presents a templated HLS-based implementation of BCAMs for exact matching and TCAMs for ternary matching, supporting arbitrary table sizes for data plane packet processing.
- It explores the trade-offs between resource and latency to flexibly cover the design space for different FPGA-based packet processing use cases.
- It leverages FPGA parallelism through applying varied design parameters and optimization directives for CAM implementation in HLS to achieve high operating frequencies with minimal latency.
- It explores the design space for CAMs by varying memory depths, key widths, and optimization methods applied to evaluate performance and resource occupation comprehensively.

The remainder of this paper is structured as follows. Section 6.2 reviews related work on FPGA-based CAM implementations. Section 6.3 introduces the potential applications in

which the proposed method can be applied practically. Section 6.4 presents the proposed HLS-based CAM design, detailing the architectural framework, optimization strategies for BCAM, TCAM, and memory allocation schemes. Section 6.5 describes the experimental environment and provides a comprehensive analysis of the results, including design space exploration, performance evaluation based on various metrics, and a comparison with State-of-the-Art approaches. Finally, Section 6.7 concludes the paper by summarizing key findings and highlighting potential directions for future research.

6.2 Literature Review

Extensive research has been conducted to optimize FPGA-based CAM architectures, focusing on improving throughput, reducing power consumption, and enhancing resource efficiency. Early implementations, such as the Hybrid Partitioned TCAM (HP-TCAM) [38], introduced partitioning techniques to mitigate the exponential growth in resource requirements associated with brute-force approaches. Although HP-TCAM improved memory efficiency, it required pre-processed, ordered data and lacked dynamic update capabilities. Subsequent designs, such as Ultra-Efficient TCAM (UE-TCAM) [39], further optimized resource utilization by reducing redundant memory structures, even though these designs remained reliant on complex Register-Transfer Level (RTL) methodologies. An FPGA-based method for the construction of deep and narrow BCAMs using Segmented Transposed Indicators RAM (STI-RAM) is proposed in [40]. This approach stores segment indicators and patterns in separate RAM structures, enabling a two-cycle match operation with concurrent reading. The authors in [41] introduce FMU-BiCAM algorithms for binary CAMs on FPGA, achieving two-cycle updates by directly using CAM keys as addresses. It reduces power consumption by eliminating lookup tables and limiting updates to necessary SRAM blocks.

LUTRAM-based CAM architectures, such as DURE-TCAM [42], D-TCAM [43], and ME-TCAM [44], have demonstrated improved update efficiency and reduced latency. DURE-TCAM [42] introduced a dynamically updateable TCAM architecture for FPGAs that uses LUTRAM primitives. DURE addresses the issue of blocking updates in existing SRAM-based TCAMs by enabling simultaneous search and update operations. The D-TCAM architecture [43] uses 6-input LUTRAMs to emulate 6-bit TCAM, and combines 64 of these LUTRAMs to create a 48-byte D-CAM block. These D-CAM blocks are cascaded horizontally and vertically to increase the width and depth of the TCAM, respectively. It also exploits the LUT-FF pair nature of FPGAs, using redundant flip-flops as pipeline registers.

ME-TCAM [44] introduces a Memory-Efficient Ternary Content Addressable Memory scheme using multipumping-enabled LUTRAM on FPGAs to enhance memory efficiency over tradi-

tional SRAM-based TCAMs. The TCAM table is partitioned into Hybrid Partitions (HPs), with each HP simulated using four LUTRAMs. However, increasing the multipumping factor improves memory efficiency at the cost of reduced throughput. RPE-TCAM [45] proposes a reconfigurable power-efficient TCAM for FPGAs that reduces power consumption by 40% using selective bank activation. It maintains one-cycle update latency and supports BiCAM and TCAM configurations. A backup CAM (BUC) handles bank overflow, ensuring efficient memory use.

An SRAM-based TCAM architecture (REST) is proposed in [46]. It improves memory efficiency by optimizing address mapping using virtual blocks (VBs). By dividing SRAM into multiple VBs, REST increases the emulated TCAM capacity while reducing throughput. The design allows a trade-off between memory efficiency and throughput by adjusting the number of VBs. A multi-region SRAM-based TCAM architecture for longest prefix matching is presented in [47], reducing memory area usage by dividing TCAM entries into index and data fields. The index maps to SRAM addresses, while the data stores content, improving memory efficiency. Another work [48] presents power-efficient FPGA-based TCAM architecture using a segmented match line strategy to reduce dynamic power consumption. Each match line is divided into four nine-bit segments, activated sequentially to prevent unnecessary power usage.

Meanwhile, flip-flop-based designs like LH-CAM [49] and G-AETCAM [50] explored the use of FPGA slice registers to achieve low-latency, high-speed CAM operations. While these designs achieved impressive performance for small-scale implementations, they faced scalability challenges due to routing complexity and hardware overhead.

TeRa [51] introduces a ternary and range-based CAM architecture for packet classification, offering a power-efficient alternative to TCAMs. It features Ternary Match Logic (TML) with NAND-NOR encoding, a carry tree-based Range Match Logic (CTRC) for range comparisons, and Match Inversion Logic for efficient rule handling. TYPE1 is optimized for ASICs, while TYPE2 is adaptable for both ASICs and FPGAs, enhancing priority selection and memory efficiency.

A reconfigurable match table (RMT) design for FPGA-based switches has been proposed in [52], incorporating a TCAM-based implementation for flexible packet processing. The design employs a three-layer structure enabling dynamic reconfiguration of match table size and type without hardware modification. A segment crossbar facilitates PMT sharing and signal conversion, including a MAND operation for combining match lines across TCAM PMTs.

Despite these advances, existing research predominantly focuses on RTL-based design method-

ologies. While RTL offers fine-grained control over hardware resources, it is inherently time-consuming, complex, and less adaptable to evolving design requirements. HLS offers more flexibility, enables faster development, enhances design portability, and simplifies design space exploration. While CAM architectures have been extensively explored at the RTL and lower design levels, HLS-based implementations remain largely unexplored, particularly for designing match-action tables in packet processing applications.

6.3 Potential Network Applications

The proposed HLS-based CAM architecture is particularly suited for high-speed data plane operations in modern networking systems. Its flexibility and configurability enable efficient integration into various packet processing applications. Below are several use cases that illustrate how CAMs can be practically deployed in real-world networking environments.

6.3.1 Packet Classification and Filtering

CAMs are commonly used to match packet header fields (e.g., IP addresses, protocol numbers, port numbers) against a set of predefined rules. This operation is fundamental in routers, firewalls, and switches to determine the appropriate action for each packet, such as forwarding, dropping, or rate-limiting. For example, a TCAM can match the quintuple of an incoming packet (source/destination IP and port, and protocol) with a set of entries in the Access Control List (ACL), allowing multi-field classification in a single clock cycle [79].

6.3.2 Firewall and Intrusion Detection Systems (IDS)

TCAMs are ideal for rule-based packet inspection, where wildcard matching is often required. For example, a firewall can use TCAM entries to define rules such as 'allow all traffic from subnet 192.168.0.X', using ternary bits to represent masked fields. IDS applications benefit similarly by matching known attack signatures or suspicious traffic behavior patterns across various protocol headers, providing real-time threat detection [80].

6.3.3 In-Band Network Telemetry (INT)

CAMs can be used to identify specific flows that require telemetry information insertion. Upon a CAM match, INT metadata is appended to the packet to track path information (e.g., latency, hop count) through the network. This enables operators to monitor network performance and detect anomalies with minimal overhead [81].

6.3.4 Network Address Translation (NAT)

In NAT-enabled devices, CAMs can serve as lookup tables to translate between internal private IP addresses and external public addresses. By maintaining a CAM of current translation entries, the device can efficiently identify and rewrite packet headers during ingress and egress processing [82].

6.3.5 Time-Sensitive Networking (TSN)

In deterministic networking environments, such as industrial automation or autonomous systems, CAMs help enforce strict scheduling and traffic-shaping rules. By matching packets against timing-based policies, CAMs enable precise control of transmission schedules to meet latency and reliability requirements [83].

6.3.6 User Management in 5G Core

CAMs play a crucial role in user session management and packet forwarding in 5G core network components, such as the User Plane Function (UPF) [84] and Access Gateway Function (AGF) [11]. In these 5G core sections, CAMs are used to match packet headers against subscriber session entries to enforce policy rules, quality-of-service (QoS) settings, and forwarding paths. This enables real-time user plane operations, including traffic steering, usage reporting, and IP anchor point management, all of which demand low-latency, high-throughput processing.

6.4 CAM Architectures, Implementations, and Trade-Offs

This work presents a templated HLS-based design and implementation of BCAM and TCAM architectures, adjusted for data plane packet processing applications on FPGAs. Our approach leverages the Vitis HLS tool for AMD (Xilinx) FPGAs, focusing on three distinct design perspectives: **Brute Force (BF)**, **Balanced (BL)**, and **High-Speed (HS)**. Each perspective addresses specific trade-offs between resource utilization, latency, and throughput, achieved by fine-tuning hardware directives provided by the Vitis HLS tool. Various design directives are applied to configure memory space allocation and access patterns based on application requirements to manage FPGA parallelism precisely. Additionally, the implementation is structured to allow the synthesis tool to optimize timing constraints and reach high operating frequencies.

6.4.1 Design Perspectives

Brute-Force CAM

The Brute-Force method implements sequential memory access, processing lookups two entries at a time. Figure 6.2 illustrates the hardware structure of the proposed CAM lookup in BF mode. The search engine is responsible for sequentially reading each entry and executing the required comparison and matching operations, depending on the CAM type. The allocated memory space consists of a single memory module with two ports for the entire CAM table. The first port is dedicated to read-only operations, while the second port supports both read and write functions.

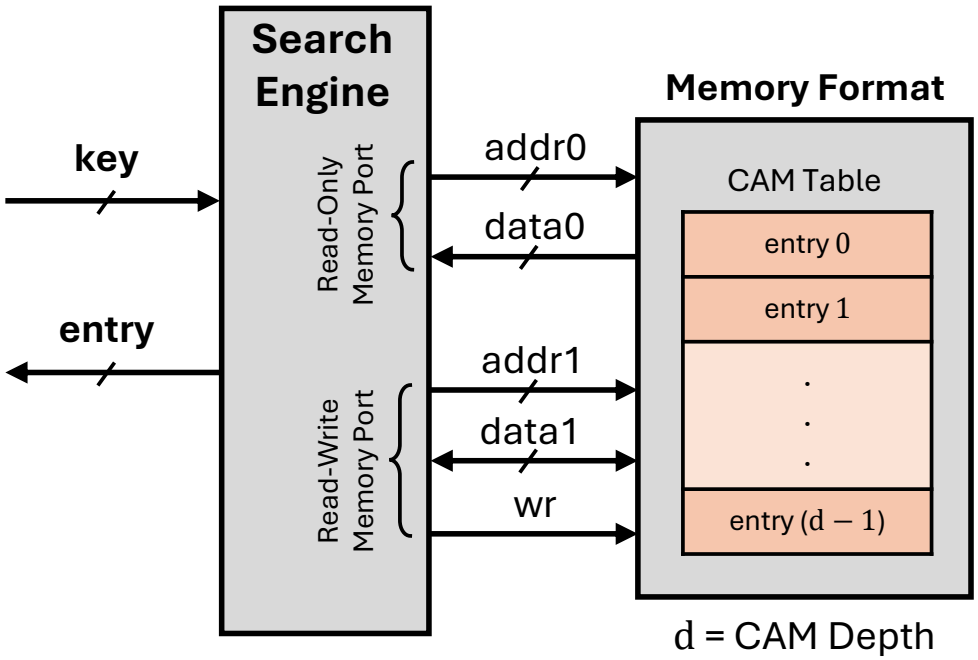

Figure 6.2 The hardware structure of CAM lookup in the BF mode on FPGA.

This approach prioritizes minimal resource utilization, making it ideal for applications with strict hardware limitations. Although it greatly reduces the consumption of FPGA resources, it introduces higher latency because of the restricted number of memory ports and the resulting serialized search procedure.

Balanced CAM

The Balanced method aims to provide a trade-off between resource utilization and performance. This architecture employs partial parallelism in memory access, where search opera-

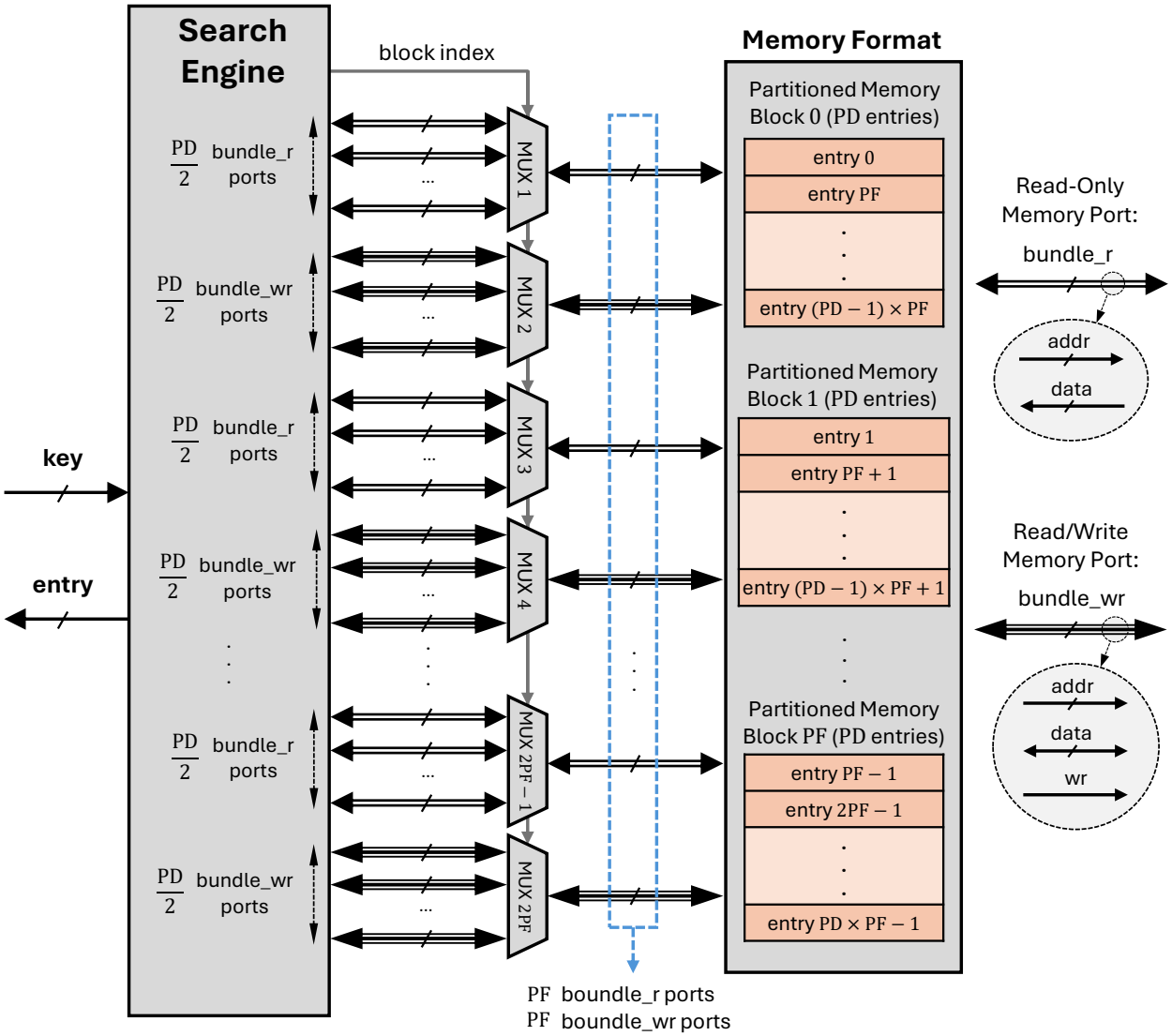

Figure 6.3 The hardware structure of CAM lookup in the BL mode on FPGA.

tions are distributed across multiple parallel memory units. Figure 6.3 depicts the proposed CAM lookup hardware structure in BL mode. Consequently, the memory is divided into PF (Partition Factor) blocks, each containing PD (Partition Depth) entries. The relationship between PF and PD is defined by Equation (6.1).

$$\textit{Partition Depth}\ (\mathrm{PD}) = \frac{\textit{DEPTH}}{\textit{Partition Factor}\ (\mathrm{PF})} \tag{6.1}$$

This partitioning scheme increases the number of memory ports by a factor of PF, resulting in 2 × PF ports. Consequently, the search engine requires PD clock cycles to read all entries. Thus, the minimum achievable latency in the BL method is PD cycles, excluding the

additional processing time required to identify the correct matching entry. During operation, the search engine schedules entry addresses and selects the appropriate address at each clock cycle using multiplexers, retrieving the entries for the matching process. The memory port signals are organized into two bundles: *bundle_r* for read-only ports and *bundle_wr* for read/write ports.

This approach reduces latency compared to the BF method while maintaining moderate resource usage. In Vitis HLS, we achieved this balance by selectively applying partial memory block partitioning, loop unrolling, and pipelining directives, allowing controlled parallel data paths. This design is well-suited for scenarios where both performance and resource efficiency are critical, such as mid-range packet processing applications.

High-Speed CAM

The High-Speed method is optimized for maximum throughput and minimal latency, fully exploiting the parallelism of FPGAs to enable simultaneous comparisons across multiple CAM entries. As illustrated in Figure 6.4, the hardware architecture in High-Speed mode features a flattened memory structure, where each entry is assigned a dedicated memory port. This configuration allows the search engine to access all entries within a single clock cycle, achieving the lowest possible latency compared to other modes. We extensively utilized directives such as full loop unrolling, aggressive pipelining, and array partitioning in Vitis HLS to maximize data flow concurrency. While this approach results in higher resource utilization, it achieves the lowest possible latency.

Given the parallel nature of operations in the HS mode, the intermediate wiring and routing logic consume significant FPGA resources. We introduce an enhanced version of the HS mode, referred to as HS-H, to improve resource efficiency. The HS-H architecture follows a hierarchical design. It matches the input key and returns the index of each matching entry, or an invalid value if no match is found. The valid indexes are then forwarded through tree-based multiplexers. This hierarchical approach reduces the number of intermediate signals propagating across multi-cycle matching operations, thereby saving logic resources compared to the original HS mode. However, the trade-off shows a slight increase in the required number of clock cycles due to the added hierarchical structure.

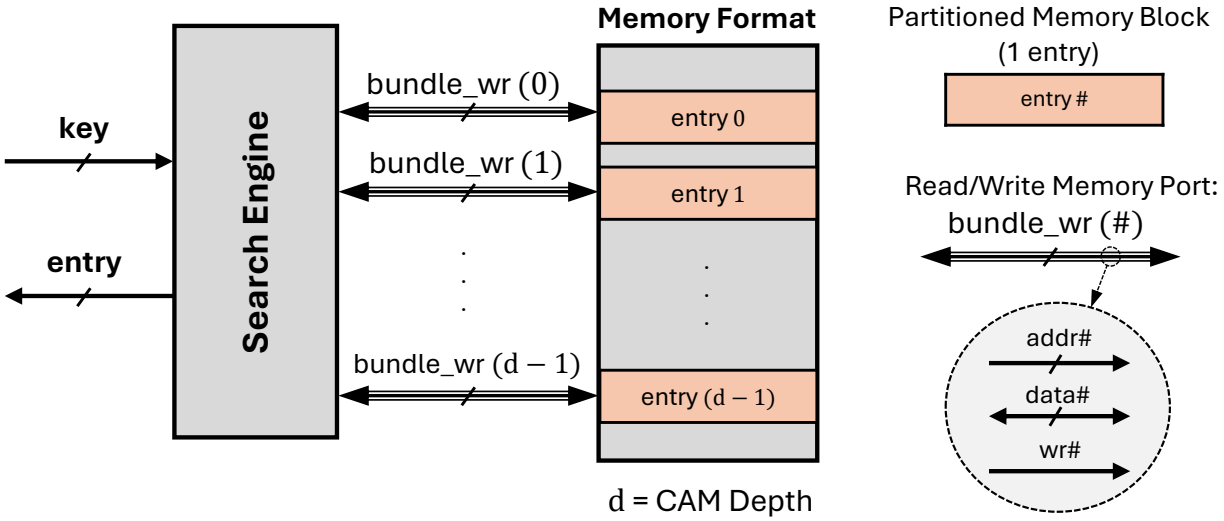

Figure 6.4 The hardware structure of CAM lookup in the HS mode on FPGA.

6.4.2 BCAM

The BCAM implementation follows a template-based structure, allowing for the flexible parameterization of key width, value width, and depth. The core structure of BCAM consists of an array of entries, where each entry is defined by a key, a corresponding value, and a validity flag as shown in Figure 6.5a. The BCAM is implemented as a class template (BCAM<KEY_WIDTH, VALUE_WIDTH, DEPTH>) to support a customizable key size, value size, and memory depth.

The `search_entry` function performs an exact match lookup by iterating through all entries, with its level of parallelism determined by the selected optimization mode. In HS mode, full parallelism is applied by distributing over distinct memory blocks, allowing all entries to be checked simultaneously for minimal latency. In BL mode, the degree of parallelism is proportionally determined by the PF, as higher PF values enable more memory partitions to be accessed concurrently, enhancing parallel lookup. In BF mode, the search is fully sequential, processing one entry at a time without parallelism. The function finally finds the matched index and returns the corresponding entry.

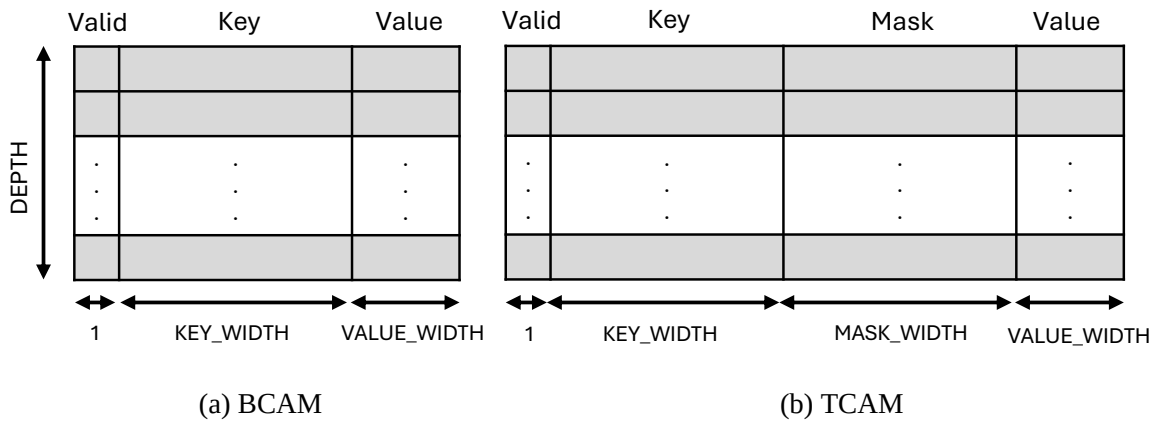

Figure 6.5 The implemented structures of BCAM and TCAM.

6.4.3 TCAM

Ternary matching in TCAM allows each bit of a stored key to represent '0', '1', or a wildcard ('X'). This is implemented by associating each key bit with a corresponding mask bit. When a mask bit is enabled, the associated key bit becomes a wildcard, matching both '0' and '1' during lookup. For example, a stored IPv4 prefix of 192.168.1.X ($C0.A8.01.XX$) can be represented by a key and a mask where the last six bits are masked, allowing any value in those positions. This would match IP addresses such as 192.168.1.10 ($C0.A8.01.0A$) or 192.168.1.63 ($C0.A8.01.3F$).

The proposed TCAM class provides a templated key-value storage system that supports ternary matching, allowing wildcard bits in key comparisons. Each entry consists of a key, a mask, a value, and a validity flag, as shown in Figure 6.5b. The `search_entry` function performs a lookup by iterating over all entries and checking for a ternary match. As illustrated in Figure 6.6, the matching process occurs in two stages. In the first stage, the function applies a bitwise XNOR between the stored and input keys to determine bit similarity. The result is then masked using the stored mask, and a validity check is performed. The outcome is stored as a Boolean value in an array, indicating whether each entry is a potential match. In the second stage, the Parallel Valid Index Finder (PVIF) module scans the array to locate the first valid match and retrieves the corresponding index and entry.

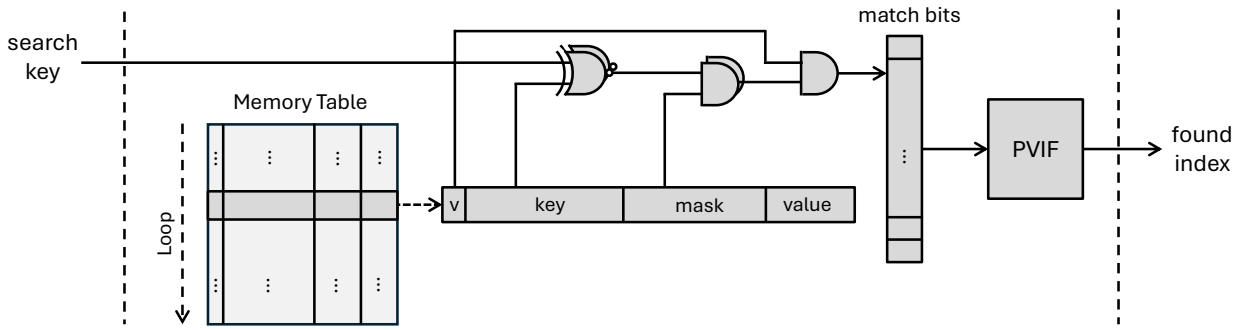

Figure 6.6 The TCAM search entry logic.

6.4.4 Memory Allocation

The proposed HLSCAM solution employs different memory allocation strategies in BF, BL, and HS modes, optimizing the use of FPGA resources based on performance and parallelism requirements. The primary memory resource utilized in the BF and the BL modes is LUTRAM, while the HS mode relies on register-based storage to maximize parallel access.

In the HS mode, the CAM array is fully flattened and mapped into FPGA registers, enabling simultaneous access to all entries. This approach eliminates the sequential memory access bottleneck associated with LUTRAM, allowing each lookup to be processed in a single cycle. However, storing CAM entries in registers instead of LUTRAM significantly increases FF utilization, which may become a limiting factor in large-scale CAM deployments. Each bit of a CAM entry in HS mode is mapped to a dedicated FF, allowing true parallel access and one-cycle lookup latency. This direct mapping ensures that all entries are immediately accessible without sequential memory traversal. However, such register-based implementation requires individual addressing for each bit, which can introduce additional logic overhead for managing lookup control and can complicate placement and routing during synthesis. This trade-off between speed and resource consumption must be carefully balanced, particularly for large-scale CAMs.

In the BF and BL modes, the CAM architecture utilizes LUT6 (LUT 6-input) elements configured as LUTRAM, where each LUT6 can store 64 bits. Figure 6.7 illustrates the LUT6 organization for a BCAM depth of 64 entries, each with a 25-bit width (1-bit valid flag, 16-bit key, and 8-bit value) in the BF mode. To accommodate the dual-port memory access used in BF mode, the entries are divided into two groups:

- **Even-indexed entries:** Mapped to the left group of LUTs.

- **Odd-indexed entries:** Mapped to the right group of LUTs.

Each group contains 32 entries, and all LUTs within a group share the same address input to provide simultaneous access. Each bit of an entry is stored at the same LUT address across different LUTs. Given the 25-bit entry width, 25 LUTs are required per group, resulting in a total of 50 LUTs for this BCAM configuration. However, due to the fixed 64-bit capacity per LUT, half of the bits in each LUT remain unused, leading to lower LUTRAM utilization efficiency. If the CAM architecture were configured with a single memory port instead of two, the LUT usage would be reduced by half, requiring only 25 LUTs instead of 50. However, this would come at the cost of approximately doubling the lookup latency, as each cycle could only access one entry at a time instead of two.

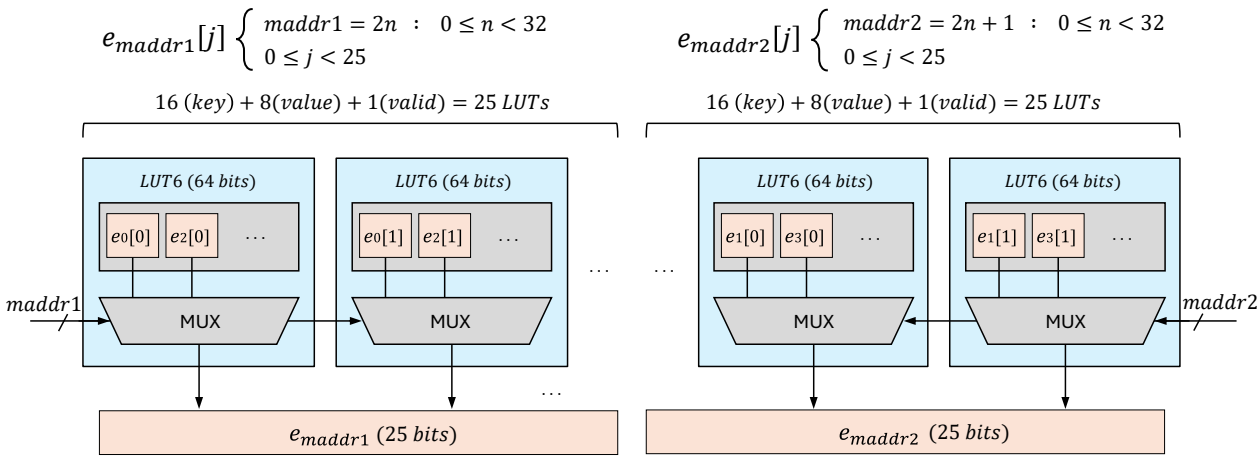

Figure 6.7 Memory allocation for the proposed BCAM in the BF mode with double memory ports and depth = 64, entry width = 25 bits (16-bit key, 8-bit value, 1-bit valid).

In this mode, the HLSCAM employs two memory ports to divide entries into two groups: even indexes and odd indexes. The even entries are mapped at the left group of LUTs, and the odd entries are mapped in the right group of LUTs, placing 32 entries into each group. The address input for the LUTs of each group is the same. Each bit of entries is distributed to the same address at LUTs. Since the entry width is 25 bits, 25 LUTs are needed for each group, occupying 50 LUTs in total. In this case, half the bits of each LUT remains unused. If we had single memory ports, the LUT usage would be half, and the latency would be around doubled.

Although double-port memory used in the BF mode provides an efficient memory structure, it suffers from limiting the available memory bandwidth, resulting in linearly growing latency by increasing the CAM depth. The CAM storage is partitioned across more than two LUTs to

enable parallel memory access, a process controlled by the PF parameter. The PF determines the number of LUTs allocated per CAM configuration, directly influencing memory access parallelism and resource utilization. Figure 6.8 shows the memory arrangement in the BL mode.

$$\begin{aligned} \text{LUTRAM usage} &= \text{ENTRY WIDTH} \times \lceil \frac{\text{CAM DEPTH}}{2^6} \rceil \times \text{\# of memory ports} \\ &= \text{ENTRY WIDTH} \times \lceil \frac{\text{CAM DEPTH}}{2^6} \rceil \times \text{PF} \times 2 \end{aligned} \tag{6.2}$$

Higher PF values allocate more LUTs to store CAM entries, increasing parallel read/write access and reducing lookup latency. However, higher PF values also lead to LUTRAM fragmentation, as each partitioned LUT may contain unused bits that cannot be reassigned to other memory segments. This results in lower LUTRAM utilization efficiency despite improved parallelism.

In the BF mode, sequential lookups require minimal partitioning, leading to higher LUT utilization efficiency but increased lookup latency due to serial memory access. In contrast, the BL mode introduces a moderate level of partitioning, striking a balance between parallel memory access and LUTRAM efficiency.

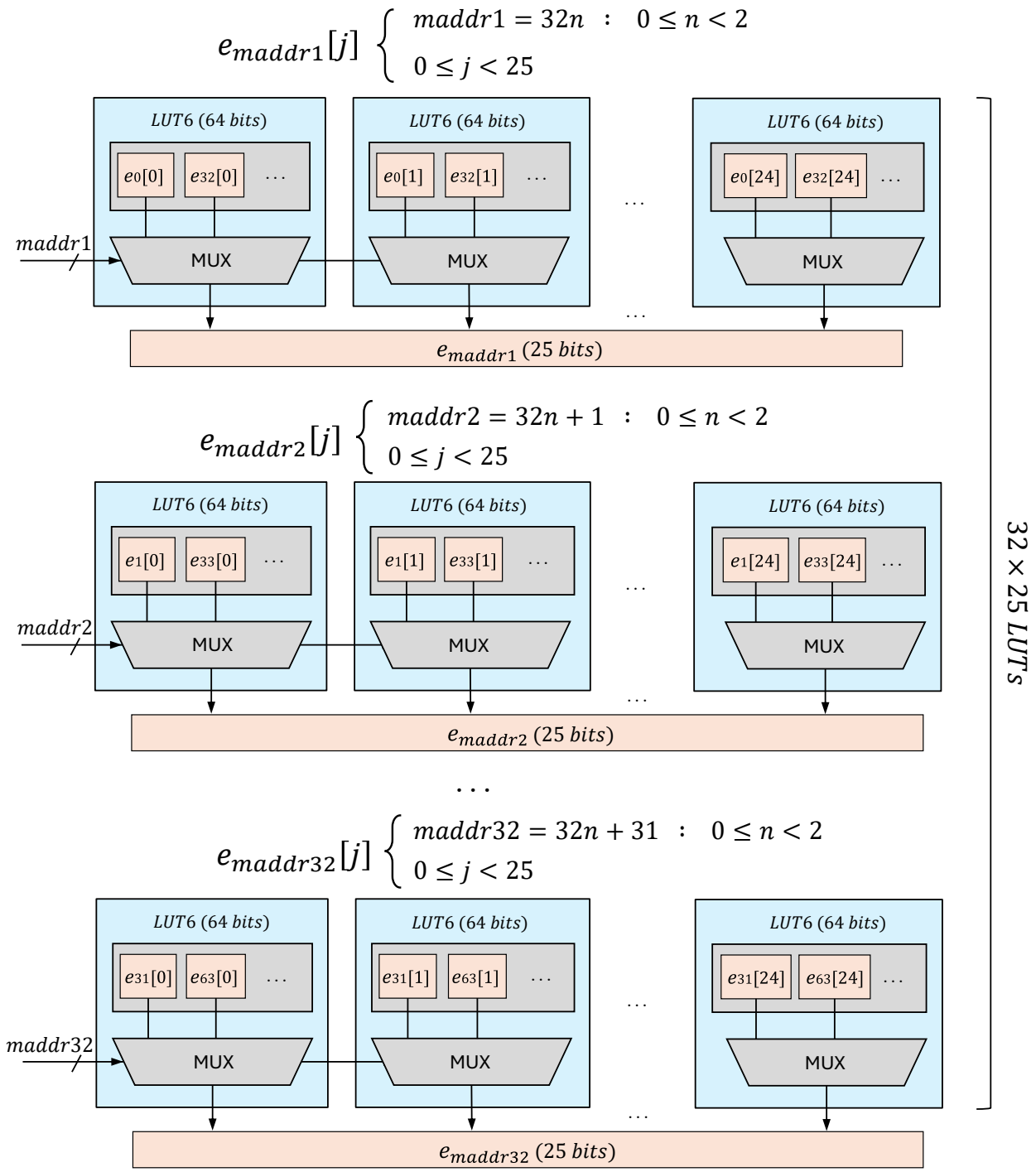

Figure 6.8 Memory allocation for the proposed BCAM in the BL mode with $PF = 16$, depth = 64, entry width = 25 bits (16-bit key, 8-bit value, 1-bit valid).

6.5 Evaluation

This section evaluates the HLS-based BCAM and TCAM implementations under various design spaces. The evaluation focuses on measuring latency, maximum frequency, throughput, and FPGA resource utilization, highlighting the trade-offs between performance and hardware efficiency. Latency was measured in terms of the number of clock cycles required for a single lookup operation. At the same time, throughput was computed based on the operating frequency and parallelism in memory access.

6.5.1 Experimental Setup

The BCAM and TCAM architectures were implemented using AMD Vitis HLS 2023.2, which enables high-level hardware description and optimization for FPGAs. The synthesized designs were evaluated using AMD Vivado to collect detailed resource utilization reports. The AMD Alveo U280 FPGA (xcu280-fsvh2892-2L-e) was targeted as a high-performance acceleration card well-suited for high-throughput networking applications. The initial clock frequency was set to 312.5 MHz to align with standard high-speed networking requirements, such as those in 10G and 25G Ethernet systems. The maximum operating frequency (f_{max}) was determined based on the critical path identified during FPGA synthesis and post-place-and-route analysis. All reported results were obtained after placement and routing to ensure accurate performance evaluation.

6.5.2 Design Space Exploration

The design space of the proposed CAM architectures was explored by varying CAM depth, key width, and optimization methods. The evaluation considered three different table depths: 64, 128, and 256 entries. The 32, 64, and 128 key widths were chosen to analyze the impact of various configurations on performance and resource utilization. The stored value word width remained constant, as the complexity of the matching process depends mostly on the key rather than the stored value. Moreover, the PF was fixed to 16 to divide CAMs into 16 partitions and provide 32 distinct memory ports. These variable parameters were selected to restrict the design space for focused analysis. At the same time, the proposed templated HLSCAM can go beyond these parameters that we evaluate with higher parameters in the Section 6.5.5. This exploration allowed us to identify optimal configurations that balance efficiency, scalability, and application-specific requirements.

The synthesis performance and utilization of resources of the proposed BCAM architecture were evaluated in different configurations, as presented in Table 6.1. In the BF mode, both

LUT and register usage increase proportionally with growing table depth and key width. Latency directly correlates with CAM depth, as each lookup requires sequential comparisons. The measured latencies are 33, 65, and 129 clock cycles for 64, 128, and 256 entries, respectively. The maximum operating frequency varies between 346 MHz and 397 MHz, depending on key width and resource constraints.

In the BL mode, LUT Logic and LUTRAM usages grow linearly with key width but LUTRAM usage remains constant as the BCAM depth increases. This behavior results from PF being fixed at 16 across all depths. As described by Equation 6.2, the number of required LUTRAMs remains unchanged until the partitioned memory blocks can no longer accommodate additional bits, increasing LUTRAM usage. Register utilization follows a similar trend, changing slightly with depth variations but linearly increasing with key width. Depending on key width and depth, the maximum lookup latency varies from 4 to 10 clock cycles. The maximum operating frequency is variable from 306 MHz to 349 MHz, which is relatively lower than that of the BF mode.

In the HS mode, the tool shifts from LUTRAM storage to register-based storage for fully partitioned entries, reducing total LUT utilization. While LUT usage in HS mode is lower than in BL mode for depths of 64 and 128 entries, it increases significantly for 256-entry configurations due to the higher intermediate logic and wiring requirements for parallel processing. The use of registers grows more rapidly than in the BL mode, reflecting the use of registers to store BCAM entries. The lookup latency is minimized to one cycle for 64 entries with a key width of 32, while it increases to two cycles for larger configurations. The maximum operating frequency ranges from 314 MHz to 345 MHz, which are close to the BL mode.

The TCAM architecture exhibits synthesis performance trends similar to BCAM as presented in Table 6.2, with increased resource utilization due to the added complexity of ternary matching and additional storage required for mask bits. Compared to BCAM, LUT, and register usage are approximately doubled across all configurations. Latency remains identical to BCAM, maintaining 33, 65, and 129 cycles in the BF mode and ranging between 5 and 10 cycles in the BL mode. In the HS mode, lookup latency remains at one or two cycles, depending on the key width and table depth. The range of maximum operating frequencies is slightly wider than in BCAM, varying from 294 MHz to 412 MHz.

6.5.3 BCAM Versus TCAM

This subsection compares the proposed BCAM and TCAM across different optimization modes, focusing on FPGA resource utilization and maximum clock frequency. The compar-

ison is based on BCAM and TCAM architectures with depths of 128 and 256, while keeping the key width fixed at 128-bit and the value width at 8-bit. Figure 6.9 compares BCAMs and TCAMs in BF mode in terms of LUT and slice register consumptions, as well as frequency. The LUT consumption is a combination of LUT logic and LUTRAMs, which are shown in the chart separately. As expected, LUTRAM usage in TCAM is approximately twice that of BCAM, reflecting the increased memory demand for storing mask bits. In contrast, LUT logic usage is nearly identical for both architectures, with only minor differences, since both require the same logic for sequential memory iteration and matching operations. Accordingly, TCAM consumes more total LUTs than BCAM. TCAM also requires 24% more registers for 128-depth configurations and 14% more for 256-depth configurations than BCAM. More-

Table 6.1 Resource Utilization and Performance Analysis of HLSCAM BCAM in the BF, BL, and HS modes. The value width is fixed to 8-bit.

Mode	Depth	Key Width (bit)	Latency (cycle)	LUT Logic	LUTRAMs	SRs1	f_{max}(MHz)
BF	64	32	33	311	82	456	397
		64	33	318	146	521	389
		128	33	488	274	661	383
	128	32	65	527	164	828	357
		64	65	594	292	902	346
		128	65	967	548	1,061	356
	256	32	129	1,025	328	1,574	373
		64	129	1,097	584	1,657	360
		128	129	1,563	1,096	1,852	364
BL	64	32	4	951	1,312	1,619	341
		64	4	1,552	2,336	2,692	334
		128	5	2,769	4,384	4,771	322
	128	32	6	1,142	1,312	1,675	349
		64	6	1,798	2,336	2,719	345
		128	7	2,952	4,384	5,074	325
	256	32	10	1,289	1,312	1,776	338
		64	10	2,092	2,336	3,086	326
		128	10	3,136	4,384	5,582	306
HS	64	32	1	1,229	0	2,627	345
		64	2	1,934	0	4,759	344
		128	2	3,267	0	8,940	337
	128	32	2	2,497	0	5,308	329
		64	2	3,899	0	9,472	316
		128	2	6,561	0	17,746	314
	256	32	2	4,992	0	10,574	317
		64	2	7,717	0	18,873	321
		128	2	12,997	0	35,386	320

over, BCAM achieves slightly higher maximum operating frequencies, surpassing TCAM by approximately 3% in the 128-depth case and 8% in the 256-depth case.

Table 6.2 Resource Utilization and Performance Analysis of HLSCAM TCAM in the BF, BL, and HS modes. A TCAM entry holds key and mask portions having the same length. The value width is fixed to 8-bit.

Mode	Depth	Key Width (bit)	Latency (cycle)	LUT Logic	LUTRAMs	SRs1	f_{max} (MHz)
BF	64	32	33	303	146	520	412
		64	33	373	274	649	404
		128	33	626	530	917	372
	128	32	65	566	292	892	386
		64	65	627	548	1,030	382
		128	65	1,114	1,060	1,317	346
	256	32	129	1,100	584	1,636	337
		64	129	1,140	1,096	1,783	340
		128	129	2,046	2,120	2,108	336
BL	64	32	4	1,563	2,336	2,644	395
		64	5	2,921	4,384	4,741	360
		128	5	5,633	8,480	8,945	357
	128	32	6	1,971	2,336	2,691	343
		64	7	3,368	4,384	4,894	341
		128	7	6,128	8,480	9,296	328
	256	32	10	2,138	2,336	2,871	335
		64	10	3,289	4,384	5,122	377
		128	10	6,197	8,480	9,565	315
HS	64	32	1	2,552	0	4,676	365
		64	2	3,361	0	8,794	346
		128	2	5,985	0	17,004	339
	128	32	2	4,437	0	9,371	333
		64	2	10,287	0	17,589	324
		128	2	11,904	0	34,000	320
	256	32	2	11,371	0	18,740	325
		64	2	17,356	0	35,183	294
		128	2	24,109	0	67,996	304

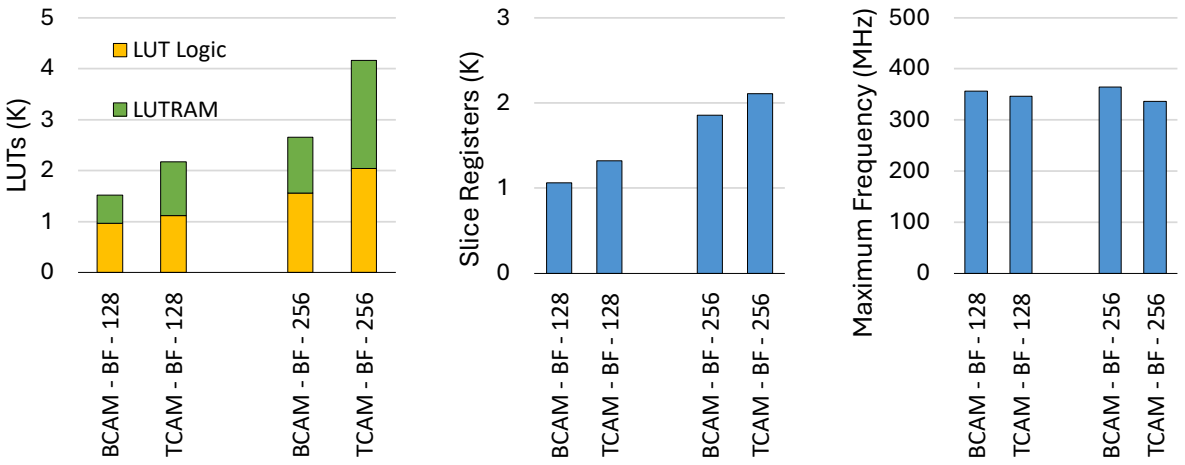

Figure 6.9 Comparison of the proposed BCAM and TCAM in the mode BF for 128 and 256 CAM depth (128-bit key, 8-bit value, 1-bit valid).

In the BL mode, the resource utilization gap between BCAM and TCAM becomes more pronounced, as shown in Figure 6.10. While LUTRAM usage follows the same trend as in the BF mode, LUT logic consumption in TCAM is approximately 98% and 108% higher than in BCAM for depths of 128 and 256, respectively. This increase is attributed to the semi-parallel processing approach in the BL mode, which requires additional logic to handle and schedule the partitioned block memory units. Registers are also increasingly demanded for the TCAM to store more and wider intermediate signals in the pipeline. Despite these differences, the maximum operating frequencies of both architectures remain within a similar range.

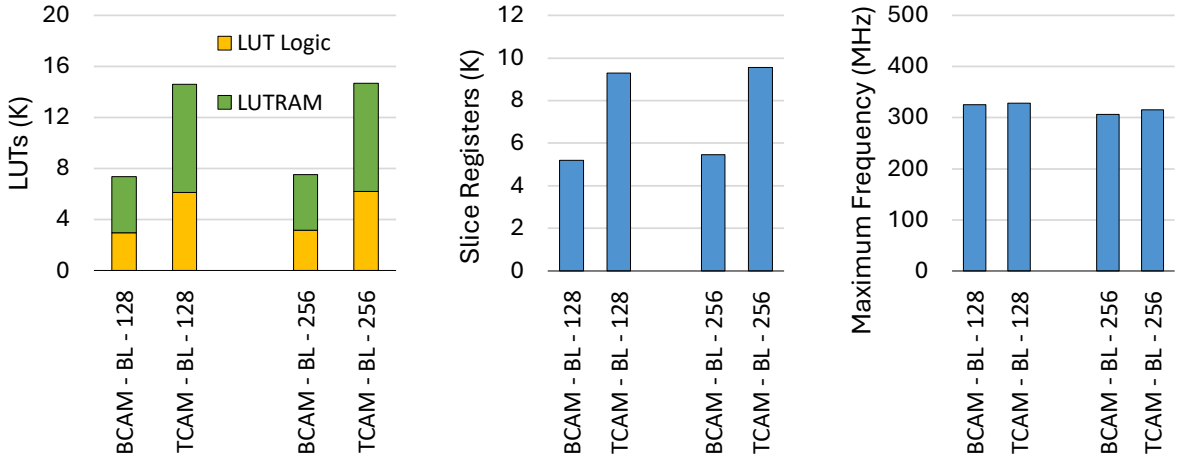

Figure 6.10 Comparison of the proposed BCAM and TCAM in the mode BL for 128 and 256 CAM depth (128-bit key, 8-bit value, 1-bit valid).

Figure 6.11 presents the comparison in HS mode, where LUTRAM is no longer used, making registers the primary resource for storage. The LUT logic and register utilization trends in HS mode follow the same pattern as in BL mode, but with higher overall values due to the increased parallelism and partitioning required for high-speed operation. The maximum clock frequency follows a similar trend to BL mode, varying from 300 MHz to 320 MHz.

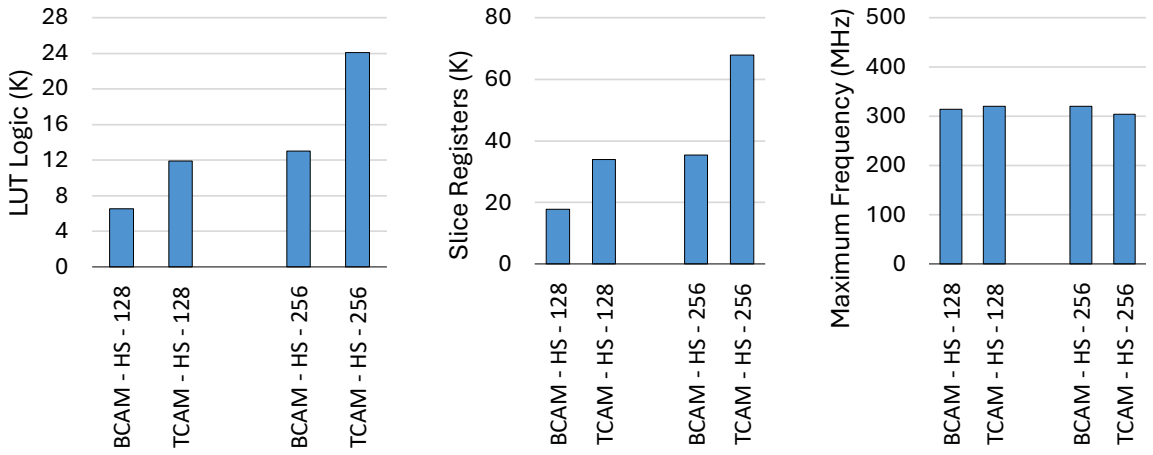

Figure 6.11 Comparison of the proposed BCAM and TCAM in the mode HS for 128 and 256 CAM depth (128-bit key, 8-bit value, 1-bit valid).

6.5.4 High-Speed (HS) Optimization Mode Variations

Furthermore, we compare the HS mode and its enhanced version, HS-H mode, across different CAM depths for the proposed TCAM design. Figure 6.12 presents a detailed comparison of resource utilization, latency, and maximum clock frequency for both modes, using TCAM configurations with depths of 64, 128, and 256, while keeping the key width fixed at 128 bits. The results demonstrate that the HS-H mode achieves up to 7% higher efficiency in LUT utilization and up to 4% higher efficiency in register utilization compared to the HS mode. The most significant improvement is observed in the maximum clock frequency, which increases by up to 32.9% for a TCAM depth of 256. However, as expected, the latency increases by one cycle when the TCAM depth is 256.

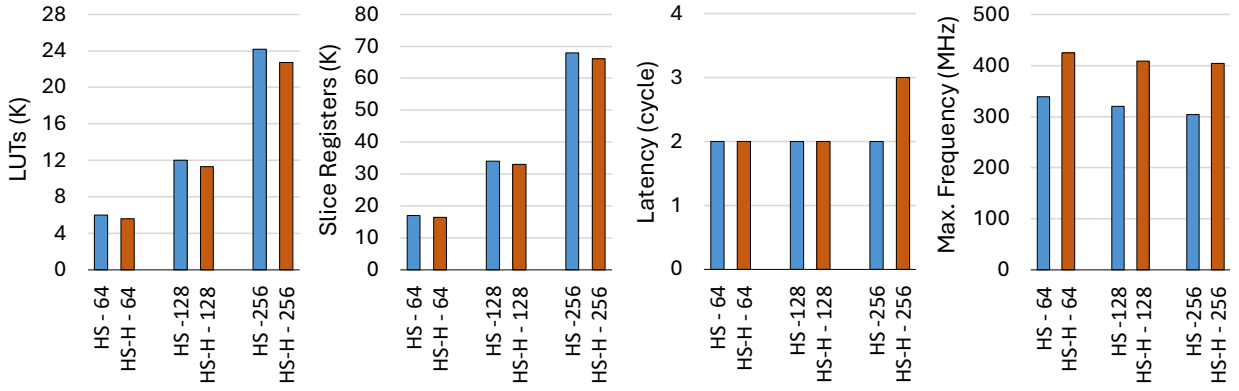

Figure 6.12 Comparison of the HS and HS-H modes for the proposed TCAM (128-bit key, 8-bit value, 1-bit valid).

6.5.5 Comparison

To evaluate the proposed HLSCAM architectures, we consider the HS-H mode, which offers improved resource efficiency and higher operating frequencies compared to the HS mode. The comparison considers key performance metrics, including FPGA resource utilization, which accounts for the combination of LUTRAMs, LUT logic, slice registers, maximum clock frequency, and throughput. Throughput TP is defined as the rate of lookup operations completed per second, which is measured in bits per second (bps) and calculated using Equation (6.3) [42]:

$$Throughput = f_{max} \times W_{CAM} \tag{6.3}$$

where the f_{max} is the maximum frequency and W_{CAM} is the key width of the CAM. Since CAM architectures in the literature are implemented on different FPGA devices, a normalized throughput (N.TP) metric is introduced to allow for a fair comparison. It is defined in Equation (6.4) [42]:

$$Normalized\ Throughput = Scaling\ Factor \times Throughput \tag{6.4}$$

where *Scaling Factor* is a coefficient that compensates for differences in FPGA technology speed, making performance metrics more comparable across different devices. The *Scaling Factor* is computed using Equation (6.5) [42]:

$$Scaling\ Factor = \frac{Technology(nm)}{40(nm)} \times \frac{1.0}{VDD} \tag{6.5}$$

where $Technology$ refers to the FPGA CMOS technology size and the VDD refers to the corresponding supply voltage. The Virtex-6 family is used as the baseline FPGA, as it is fabricated with a 40 nm process and operates at 1.0 V supply voltage. Table 6.3 lists the related FPGA parameters used to calculate the normalized throughput. It provides the relevant FPGA parameters, including those for Virtex-6, Kintex-7, Virtex-7, and Virtex Ultrascale+, enabling the calculation of scaling factors for each. The comparison results are compiled to reflect post-place-and-route implementations.

Table 6.3 Different FPGA parameters

FPGA Family	CMOS Technology	VDD	Scaling Factor
Virtex-6	40 nm	1.0 V	1.0
Virtex-7, Kintex-7	28 nm	1.0 V	0.7
Virtex Ultrascale+	16 nm	0.85 V	0.47

Table 6.4 shows the results of the proposed HLSCAM architectures in the HS-H mode and other existing work. The comparison results are obtained after post-place-and-route implementation for both Virtex-7 (xc7v2000tflg1925-2) and Virtex Ultrascale+ (xcu280-fsvh2892-2l-e). Resource usage is reported in terms of LUTs, FFs, and BRAMs. Additionally, CLBs (Configurable Logic Blocks) are used for Virtex UltraScale+ device, while Slices are reported for older FPGA families. The TP and $N.TP$ represent the throughput and normalized throughput, respectively. However, a direct quantitative comparison is still challenging, as not all works report the same set of resource utilization metrics, as well as variations in FPGA devices, synthesis tools, and experimental setups. Additionally, since each FPGA slice contains multiple LUTs and registers, comparing slices alone does not fully reflect differences in resource efficiency across different architectures. Despite these challenges, a qualitative comparison is made by analyzing key performance characteristics.

Some existing TCAM implementations, such as Scalable TCAM and Zi-TCAM, report high slice usage, whereas the proposed architectures demonstrate more efficient resource allocation while achieving significantly higher operating frequencies. Unlike Z-TCAM and UE-TCAM, which rely on BRAM-based storage, the proposed architectures avoid BRAM dependency, improving scalability for high-speed large CAM sizes. Since the HLSCAM uses high-level partitioning, particularly in the HS mode, using BRAM would lead to underutilized memory blocks.

The proposed HLSCAM architectures achieve higher maximum operating frequencies than most existing designs, such as Hierarchical TCAM (109 MHz), Zi-TCAM (38 MHz), and HP-TCAM (118 MHz). Compared to G-AETCAM, which operates at 358 MHz, the proposed

512 × 36 TCAM architecture sustains competitive clock speeds while handling larger key widths and CAM depths. Compared to D-TAM, the corresponding HLSCAM configuration exhibits slightly lower operating frequency, resulting in lower throughput. Nevertheless, in the wider CAMs such as 256 × 128, the HLSCAM achieves up to 31.18 Gbps normalized throughput. For similar CAM sizes, HLSCAM achieves higher operating frequencies than Xilinx CAM IPs. For TCAM 256 × 128, HLSCAM runs at 404 MHz versus 316 MHz. For

Table 6.4 Resource Utilization and Throughput Comparison with Existing FPGA-based CAM Architectures

Work	CAM Size ($D \times W$)	Device	LUTs	SRs1	BRAMs (18k/36k)	Slices/ CLBs 2	f_{max} (MHz)	TP (Gbps)	N. TP 3 (Gbps)
HP-TCAM [38]	512 × 36	Virtex-6	6,546	2,670	0/56	NA	118	4.25	4.25
UE-TCAM [39]	512 × 36	Virtex-6	3,652	1,758	0/32	NA	202	7.27	7.27
FMU-BiCAM [41]	512 × 36	Virtex-6	1,257	732	64/0	NA	284	10.22	10.22
DURE [42]	512 × 36	Virtex-6	NA	NA	0/0	1,668	335	12.06	12.06
D-TCAM [43]	512 × 36	Virtex-6	NA	NA	0/0	968	460	16.56	16.56
REST [46]	72 × 28	Kintex-7	130	390	0/1	77	50	1.40	0.98
LH-CAM [49]	64 × 36	Virtex-6	1,259	70	0/0	NA	340	12.24	12.24
G-AETCAM [50]	64 × 36	Virtex-6	3,067	4,672	0/0	NA	358	12.89	12.89
Sca. TCAM [85]	1024 × 150	Virtex-7	NA	37,556	0/0	20,526	199	29.85	20.9
Hie. TCAM [86]	512 × 72	Virtex-6	16,720	NA	0/0	NA	109	7.85	7.85
Zi-TCAM [87]	512 × 36	Virtex-6	24,551	NA	0/0	NA	38	1.37	1.37
Z-TCAM [88]	512 × 36	Virtex-6	NA	NA	0/40	1,116	159	5.72	5.72
Mul. TCAM [89]	512 × 32	Virtex-6	1,515	1,593	0/16	NA	237	7.58	7.58
Xilinx BCAM [90]	256 × 128	Virtex-US+	2,583	4,159	0/6	641	383	49.04	23.05
Xilinx BCAM [90]	512 × 36	Virtex-US+	1,972	4,159	0/2	460	400	14.41	6.77
Xilinx BCAM [90]	1024 × 150	Virtex-US+	2,950	4,488	0/12	723	333	49.95	23.48
Xilinx TCAM [91]	256 × 128	Virtex-US+	15,669	19,033	0/48	3,361	316	40.51	19.04
Xilinx TCAM [91]	512 × 36	Virtex-US+	19,033	12,407	0/24	2,332	323	11.63	5.47
Xilinx TCAM [91]	1024 × 150	Virtex-US+	21,113	25,900	0/60	4,873	320	47.98	22.58
HLSCAM BCAM (proposed)	256 × 128	Virtex-7	11,786	33,776	0/0	8,410	335	42.88	30.02
	512 × 36	Virtex-7	8,138	19,722	0/0	5,872	358	12.91	9.04
	1024 × 150	Virtex-7	54,735	157,331	0/0	34,169	298	44.74	31.32
	256 × 128	Virtex-US+	12,841	33,297	0/0	3,933	423	54.10	25.43
	512 × 36	Virtex-US+	7,194	19,461	0/0	2,402	392	14.11	6.63
	1024 × 150	Virtex-US+	54,572	155,748	0/0	15,881	374	56.03	26.34
HLSCAM TCAM (proposed)	256 × 128	Virtex-7	22,724	66,329	0/0	14,719	348	44.54	31.18
	512 × 36	Virtex-7	12,755	38,663	0/0	10,435	339	12.21	8.55
	1024 × 150	Virtex-7	105,055	311,431	0/0	66,213	315	47.20	33.04
	256 × 128	Virtex-US+	22,642	66,119	0/0	6,632	404	51.70	24.30
	512 × 36	Virtex-US+	13,086	38,266	0/0	5,805	365	13.12	6.12
	1024 × 150	Virtex-US+	105,147	309,613	0/0	29,766	333	49.97	23.48

TCAM 512×36, it operates at 365 MHz compared to 323 MHz. This results in greater normalized throughput 24.30 Gbps versus 19.04 Gbps and 6.12 Gbps versus 5.47 Gbps. However, Xilinx CAM IPs use BRAM for balanced resource utilization, while HLSCAM relies on distributed memory, increasing CLB usage.

Although HLSCAM TCAM and BCAM consistently achieve significantly higher maximum frequencies and related throughput TP than most existing works, the resulting normalized throughput $N.TP$ gains are not proportionally higher. This is due to the scaling factor introduced by the normalization equation, which is not ideal for accurately reflecting performance improvements across different architectures and configurations.

6.6 Discussion

The proposed HLSCAM framework introduces a flexible, high-performance CAM implementation in HLS, enabling users to define their CAM table sizes according to specific application requirements. Unlike fixed hardware designs, this adaptability allows for customized trade-offs between resource utilization and performance, making it a versatile solution for FPGA-based packet processing.

To further enhance flexibility, HLSCAM provides multiple optimization strategies, offering users a range of configurations to balance throughput, latency, and resource consumption. The three primary operational modes (BF, BL, and HS) are designed to meet diverse performance demands. The BF mode is particularly suited for resource-constrained scenarios, offering a highly compact implementation. In contrast, the HS mode prioritizes minimal latency by leveraging parallel registers at the cost of higher resource consumption. The BL mode provides a balanced trade-off between latency and resource efficiency through partitioning memory blocks into multiple LUTRAMs. The PF parameter plays a critical role in this trade-off, acting as a control knob to shift the design towards either a resource-efficient or low-latency CAM table.

Furthermore, we introduced an improved version of the HS mode, termed HS-H, which retains the fundamental parallel processing benefits of the HS mode but adopts a hierarchical structure. This architectural enhancement significantly reduces intermediate signal propagation, leading to up to 7% LUT reduction and up to 4% reduction in register utilization, while also achieving up to 32.9% improvement in maximum operating frequency.

HLSCAM is specifically tailored for dataplane packet processing, distinguishing it from traditional CAM architectures. The entry format is adapted to accommodate network rule storage, incorporating a valid bit to determine entry validity and a value field for packet

processing upon a successful match. For TCAM implementations, mask bits are explicitly defined for the entire key width, leading to higher memory consumption compared to conventional CAM solutions. As a result, for the same CAM dimensions, HLSCAM requires more than double the memory units used in existing works.

Despite its relatively higher FPGA resource consumption, HLSCAM outperforms many competitors in terms of frequency and throughput, particularly in the 512×36 configuration. Additionally, it delivers substantial throughput in configurations such as 256×128 and 1024×150. A key factor contributing to HLSCAM's resource consumption is its HLS-based implementation. Unlike HDL-based solutions, which allow fine-grained control over hardware resources, HLS-based designs introduce abstraction layers that can lead to additional overhead. While this abstraction enhances design flexibility, it often results in less efficient resource utilization than RTL implementations. This inefficiency becomes more pronounced in larger designs that rely on deep pipelining, where mapping high-level behavioral code to FPGA logic and memory introduces complexity. One way to mitigate this issue is to reduce memory partitioning, simplifying the design to lower memory and logic overhead. A hybrid approach that combines semi-partitioned memory with a hash-based lookup scheme could further optimize resource efficiency by reducing the number of required comparisons. However, potential hash conflicts must be carefully considered. Investigating the feasibility and effectiveness of these approaches, along with their impact on resource utilization and lookup performance, remains an area for future research.

Nevertheless, HLSCAM enables the easy generation of arbitrary CAM configurations, while automatic scheduling mechanisms optimize logic placement and pipelining to achieve high operating frequencies. These features are beneficial for high-speed networking and packet processing applications, including rule-based packet flow filtering, Intrusion Detection Systems (IDS), In-band Network Telemetry (INT), timestamping, and packet classification.

6.7 Conclusions

This paper presented HLSCAM, a fine-tuned HLS-based implementation of BCAM and TCAM for high-speed packet processing on FPGAs. By leveraging High-Level Synthesis, the proposed approach enables configurable, scalable, and resource-efficient CAM architectures, reducing development complexity compared to traditional RTL-based implementations. The design methodology employed different design parameters to provide a fine-tuned FPGA parallelism management. This approach, combined with a structured implementation, allowed the synthesis tool to decrease the critical path. The Brute Force (BF), Balanced (BL), and High-Speed (HS) modes offer a spectrum of design trade-offs between latency, resource

utilization, and throughput, allowing customization for different networking applications.

The evaluation results demonstrated that HLSCAM achieves significantly higher operating frequencies than most existing FPGA-based CAM designs, reaching up to 423 MHz. The HS mode achieved the lowest lookup latencies, while the BL mode provided a balance between resource efficiency and performance. Despite higher resource consumption due to extensive partitioning and parallelism, the proposed architectures delivered competitive throughput across various configurations. A comparison with previous work highlighted HLSCAM's advantages in flexibility, scalability, and high-speed performance, particularly in large-scale CAM applications. By avoiding BRAM dependency, the proposed architectures improve scalability, making them well-suited for high-performance packet classification and filtering.

Future work will further optimize HLS-based CAM architectures to save resources while maintaining rapid lookup. Approaches like hash-based methods could prevent CAMs from comparing with all entries to reduce logic and the need for memory partitioning, thereby improving efficiency. Additionally, integrating these CAMs into real-world network processing pipelines will provide further insights into their applicability in next-generation networking systems.

CHAPTER 7 ARTICLE 3: P4THLS: A TEMPLATED HLS FRAMEWORK TO AUTOMATE EFFICIENT MAPPING OF P4 DATA-PLANE APPLICATIONS TO FPGAS

M. Abbasmollaei, T. Ould-Bachir and Y. Savaria, (published in 2025-09-12). "P4THLS: A Templated HLS Framework to Automate Efficient Mapping of P4 Data-Plane Applications to FPGAs," in *IEEE Access*, vol. 13, pp. 164829-164845, 2025, doi: 10.1109/ACCESS.2025.3610893.

Abstract

The rising demand for high-bandwidth, low-latency network processing has led to a significant shift towards programmable data planes. The P4 language enables network operators to define packet processing pipelines flexibly. However, efficiently deploying P4-defined applications onto Field-Programmable Gate Arrays (FPGAs) remains a complex task due to the low-level hardware design requirements. This paper introduces P4THLS, a templated high-level synthesis framework that converts P4 data-plane applications into synthesizable C++ code for FPGA deployment. Key contributions include an automatic design process, a templated data structure and bus width, and unified memory management techniques. The proposed P4THLS architecture is evaluated through experiments, showing significant improvements in throughput, latency, and resource utilization over existing FPGA-based packet processing methods. The experiments demonstrate that P4THLS achieves up to 143.3 Gbps throughput with a 512-bit bus and 76.5 Gbps with a 256-bit bus, supports match-action tables with up to 64K entries, and sustains sub-50-cycle processing latency at 250 MHz, with end-to-end latencies of around 8 μs.

Keywords
Data plane, FPGA, High-Level Synthesis, P4 Language, Packet processing

7.1 Introduction

The exponential growth of network traffic, driven by advances in cloud computing, the Internet of Things (IoT), 5G networks, and real-time applications, has placed significant demands on modern networking infrastructure. Data plane devices responsible for fast, efficient packet forwarding are now required to perform increasingly complex operations at line rates while maintaining flexibility and adaptability [8, 92, 93]. Traditional network devices, such

as routers and switches, optimized for fixed-function processing, struggle to cope with the evolving needs of dynamic network environments, particularly in terms of scalability, resource efficiency, and programmable flexibility [94, 95].

Field-Programmable Gate Arrays (FPGAs) are emerging as a powerful alternative for enhancing data plane devices. With their ability to perform highly parallel processing and reconfigurable architecture, FPGAs support high-throughput packet processing and adapt to changing network protocols and requirements [95, 96]. FPGAs can scale logic and memory resources, making them effective for high-bandwidth tasks [31, 97–100]. Their versatility allows deployment across various FPGA platforms, from low-cost devices for educational and smaller-scale commercial use to high-end platforms for demanding industrial workloads. However, deploying FPGAs in data plane environments presents challenges. The conventional FPGA development flow is time-consuming and complex, especially when integrating software-defined networking (SDN) principles.

P4 (Protocol-Independent Packet Processing) [10] is designed to enable programmable data planes, allowing network operators to define how packets are processed based on high-level, protocol-independent descriptions. While P4 provides a flexible way to express packet-processing pipelines, the efficient mapping of these pipelines onto FPGA hardware remains a significant technical challenge [101, 102]. The inherent differences between the high-level abstractions used in P4 and the low-level, resource-constrained nature of FPGAs require careful consideration of memory management, processing throughput, and resource utilization.

Designing FPGA-based solutions for such environments traditionally requires extensive hardware expertise and is often time-consuming [32, 103]. High-level synthesis (HLS) has transformed this process by allowing developers to describe FPGA designs using high-level programming languages like C++ instead of low-level hardware description languages [17]. This abstraction improves design productivity and enables rapid iterations. Building on the benefits of the P4 language, one can further simplify the description of network data plane operations.

This paper presents a novel approach to bridge this gap by introducing a templated HLS framework that allows P4 applications to port to FPGA platforms seamlessly. The key contributions of P4THLS are as follows:

1. Automated code generation and mapping of P4 programs to human-readable HLS data structures.

2. A templated architecture supporting variable bus widths and enabling data transfer via AXI data streams.

3. A unified memory interface that allows choosing memory types based on application requirements.

4. Generation of a compatible driver for control plane functionality to manage match-action tables.

5. An open-sourced framework, accessible for the research community to further adapt, and improve[1].

The remainder of this paper is organized as follows: Section 7.2 provides a literature review of current FPGA-based data plane processing approaches. Section 7.3 describes the proposed templated HLS architecture and the associated implementation method. Section 7.4 presents experimental assessments of the proposed framework, showcasing its performance across several P4 applications. Section 7.5 discusses potential directions for extending this work, and Section 7.6 provides concluding remarks.

7.2 Literature Review and Background

7.2.1 P4 and Data Plane Packet Processing

P4 is a specialized declarative language designed to configure the data plane of network devices. Initially introduced by Bosshart et al. in 2014 [10], and refined in 2016 by Budiu and Dodd [104], leading to the creation of two versions: $P4_{14}$ and $P4_{16}$. P4 goals are reconfigurability, protocol-independence, and target-independence. These features make P4 adaptable to various hardware targets without prior knowledge of the architecture [6].

P4's flexibility has led to its widespread use in programmable network switches, built around the protocol-independent switch architecture (PISA) [10]. As depicted in Fig.7.1, the PISA architecture includes three components: a parser for extracting packet information, a match-action pipeline for processing packets based on rules, and a deparser to reassemble packets before forwarding them.

7.2.2 FPGA-Based Implementations for P4 Codes

Several frameworks based on academic research and commercial solutions have been proposed to translate P4 programs into FPGA-compatible implementations.

Commercial tools like AMD Xilinx VitisNetP4 [57] and Intel's P4 Suite [58] have advanced P4 FPGA deployment. These tools enable network operators to define packet processing

[1]The P4THLS framework is publicly available at https://github.com/abbasmollaei/P4THLS

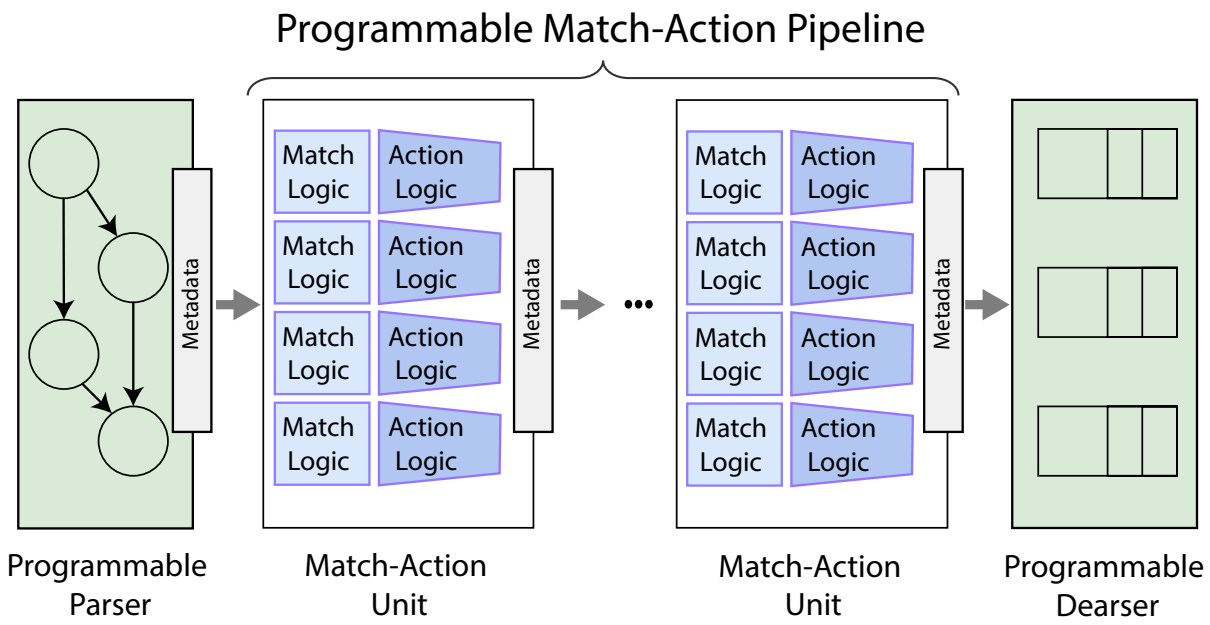

Figure 7.1 PISA architecture

in high-level P4 code, which is automatically translated into hardware description language (HDL) for FPGA deployment. Integrated with environments like Vivado and Quartus Prime, they streamline the software-to-hardware transition, reducing complexity. However, their proprietary nature limits cross-platform compatibility and portability. While effective for standard tasks, they struggle with complex or stateful functions, and integrating custom P4 externs can be cumbersome, reducing flexibility.

Various academic approaches for deploying P4 on FPGAs have been explored. The P4-to-VHDL generator [59] creates high-speed packet parsers by translating P4 into synthesizable VHDL. The P4FPGA framework [64] achieves line-rate throughput using hardware acceleration, translating P4 into Bluespec SystemVerilog and synthesizing it to Verilog. Another approach [62] introduces a protocol-independent parser for SDN networks, using a two-step P4-to-Python and Python-to-RTL process. The P4→NetFPGA workflow [63] simplifies FPGA programming for users with limited hardware expertise, leveraging the Xilinx P4-SDNet compiler and the NetFPGA SUME platform for rapid prototyping. A further study [66] proposes a match-action hardware architecture for FPGAs, generating VHDL code from P4 and optimizing resource usage. While these frameworks offer valuable insights, many, such as P4-to-VHDL [59] and P4FPGA [64], are outdated or limited. Others, like P4→NetFPGA [63], are tied to specific platforms, reducing portability and flexibility. To highlight the limitations of prior frameworks, Table 7.1 compares their core functionalities

and constraints, focusing on the aspects addressed or improved by the proposed P4THLS framework.

For the comparative evaluation in this work, we focus on two recent and representative solutions: AMD's VitisNetP4 IP [57] and the P4-to-FPGA framework [66]. These serve as the relevant baselines to assess the flexibility, performance, and resource efficiency of the proposed P4THLS framework.

7.2.3 Templated Architecture

The templated approach to converting P4 to FPGA code is not new. Previous works have proposed similar methodologies for different parts of the P4 pipeline. For instance, Jefferson's work [60] introduced a templated structure based on HLS/C++ methodology targeting the parser block. Furthermore, [66] proposed a VHDL-based templated converter that translates a full P4 program into RTL, considering the entire PISA pipeline. However, this solution is closed-source and limited in scope.

In contrast, our paper uses the templated approach to extend beyond previous works by providing an open-source, HLS-based solution that includes full bitstream generation, a driver interface, and a unified memory interface. Although the overall packet processing pipeline follows the standard PISA model, the proposed architecture introduces several innovations that distinguish it from conventional approaches. The entire pipeline is automatically generated as human-readable HLS C++ code, making it easy to customize and extend for advanced use cases. Furthermore, the design incorporates a configurable unified memory interface capable of transparently utilizing various memory sources and supports variable bus-width selection to balance latency against resource utilization. These features collectively provide a flexible, FPGA-friendly architecture that is open source and tunable, enabling developers to optimize packet processing for a wide range of applications.

7.3 P4THLS: Architecture and Implementation

7.3.1 Workflow from P4 to HLS

The P4 compilation pipelines, such as P4C [105], have three main stages: front-end, mid-end, and back-end. The front-end parses the P4 source code, validating its syntax and extracting key information like headers, parsers, and match-action tables. The mid-end optimizes this data and generates intermediate representations (IR). In P4C, the IR can be stored in a JSON file. The back-end converts these representations into hardware-specific code for target

Table 7.1 Existing P4-to-FPGA frameworks: features and limitations

Work	Features	Limitations
VitisNetP4 (AMD Xilinx) [57]	Vendor-supported, high-performance P4-to-FPGA flow, rich CAM and match-engine generation, backpressure and multi-clock support	Locked to AMD/Xilinx targets, only ultrascale+ and higher families supported; closed source
Intel's P4 Suite [58]	Automates RTL generation from P4, includes control-plane API and P4 custom architecture support	Internal-use only; no public support or documentation, limiting usability and portability.
P4FPGA [64]	Translates P4 to Bluespec SV, then to Verilog, achieving line-rate throughput	High latency; lacks modern features like variable bus widths, memory selection, or source-level editability.
P4-to-SDNet [62]	Enables P4 deployment on NetFPGA SUME using P4-SDNet tools; provides extern libraries and control APIs in Python/C	Tied to specific platform (NetFPGA SUME); requires HDL for extern implementations; limited generalizability.
P4→NetFPGA Workflow [63]	Simplifies P4 prototyping on NetFPGA SUME; supports teaching, externs, and APIs for control-plane	Platform-specific (NetFPGA SUME); not portable; requires underlying proprietary toolchain.
P4-to-FPGA [66]	Full PISA pipeline support in templated VHDL; optimized resource usage	Closed source, limited configurability; no bus-width tuning or memory abstraction interface.

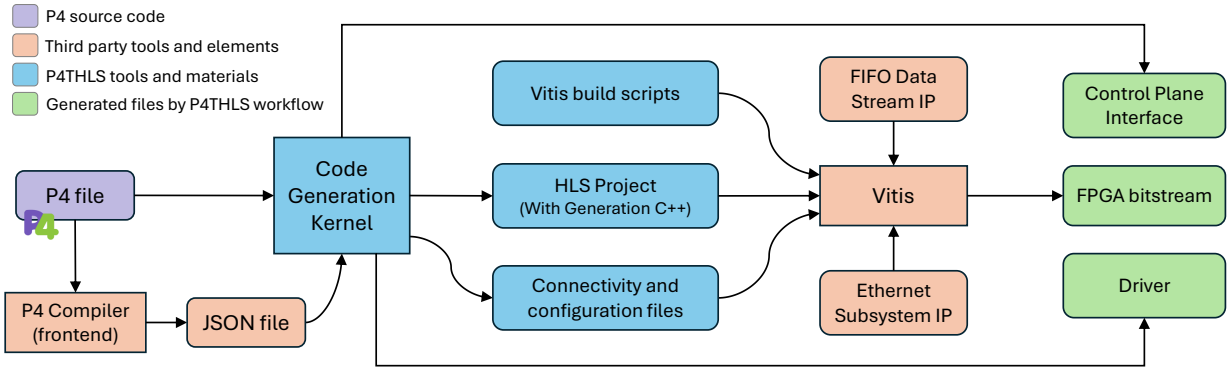

Figure 7.2 P4THLS Automation Workflow

architectures such as FPGAs or ASICs.

The P4THLS framework bridges the gap between the front/mid-end and hardware back-end. Instead of generating low-level hardware code, P4THLS extracts parameters such as table sizes, match-action logic, and headers from the mid-end, converting them into templated C++ code for the HLS compiler. Additionally, P4THLS automates the transformation of P4 programs into FPGA-executable code. Fig. 7.2 illustrates the P4THLS workflow, showing how a P4 source file produces three final outputs: a control plane interface, a driver, and an FPGA bitstream.

Code Generation Kernel

The workflow begins with the P4 source code, which is processed by a third-party P4 compiler to generate a JSON file. The first phase of the P4THLS framework involves extracting the

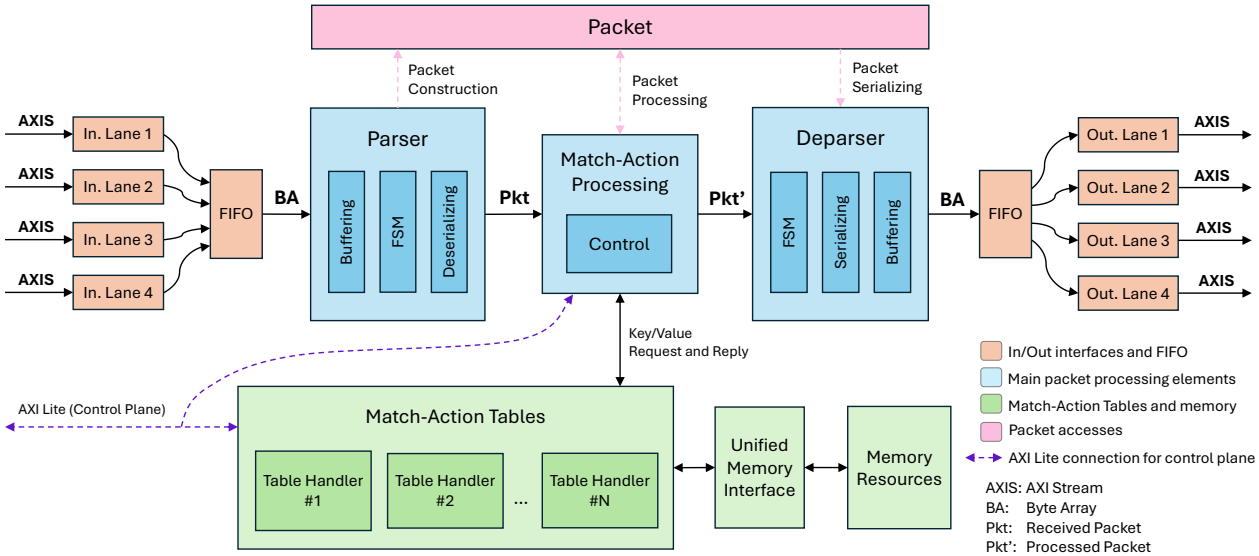

Figure 7.3 The template architecture of the HLS project generated by P4THLS

necessary parameters from this JSON file. Next, P4THLS constructs a templated HLS module based on the extracted parameters. In parallel, P4THLS generates a driver for the HLS module. This driver is built using an AXI-Lite interface, enabling the control plane to interact with the data plane core. The driver allows for real-time managing and monitoring tables, registers, counters, and other essential control elements.

The Code Generation Kernel is implemented as a Python-based tool that automates the creation of the target code. It accepts user-defined parameters to produce a fully functional HLS project. The framework employs partially templated C++ structures for the backbone of the HLS kernel, while dynamically generating code segments that depend on the given P4 program, such as header definitions, parser logic, and control flow. An example command to invoke the tool is shown below:

```
python run_p4thls switch.p4 switch.json \
        --mtype bram --bwidth 128 \
        --inport 2 --outport 2
```

This example generates an HLS kernel for the `switch.p4` application, configured to use BRAM for table storage, a 128-bit AXI interface, and two input and two output ports. The P4 program should follow the PISA architecture; otherwise, unsupported code sections are ignored. Furthermore, differences in JSON output between P4 compilers may affect compatibility. The current implementation primarily supports the BMv2 compiler (version 1.15.0) and the VitisNetP4 compiler (version 2023.2).

FPGA Bitstream Generation

After generating the HLS module and driver, P4THLS exports the IP core, which is a self-contained module that can be easily integrated into FPGA designs. This IP core is imported into an FPGA synthesis tool, such as AMD Vitis. Vitis synthesizes the HLS-generated IP, creates the FPGA configuration, and connects the IP to external interfaces, memory controllers, and networking ports. The final bitstream is then generated, ready for deployment onto the FPGA hardware.

7.3.2 Packet Processing Pipeline

Fig. 7.3 illustrates the packet processing pipeline, which consists of distinct sections responsible for specific tasks, such as parsing, processing, deparsing, and match-action processing. These elements construct a packet processing core comprising a PISA-like pipeline. Incoming and outgoing packets are sent using AXI stream interfaces.

Parser and Deparser

The parser and deparser consist of a buffering block, a finite-state machine (FSM), and serialize/deserialize functions. The deserialize function reads a byte array and reconstructs the header fields, allowing the parser to reconstruct the outgoing packets properly. The serialize function converts the header fields into a contiguous byte array.

A P4 parser inherently operates as a variable-step, sequential process, which makes its implementation complex. To simplify this for programmable logic devices such as FPGAs, the parser order must often be reformatted without altering its functionality [60]. The process starts by extracting the parser directed acyclic graph (DAG). To resolve this, we employ a leveling method to refine the parser graph, ensuring that the logic for each node is not repeated. The leveling approach constructs a leveled graph where nodes at the same distance from the starting node are grouped into the same level. This method identifies the longest distance for each node and assigns it to a level corresponding to that distance, preventing logic replication.

The internal skeleton of the parser C++ code is formed based on what Listing 7.1 shows. Consecutive switches are placed to handle the states of each level. In the case of acceptance, the constructed packet is written to the packet stream to deliver the match-action processing section.

The P4 deparser is responsible for reconstructing the packet and arranging all modified

Listing 7.1 Parser code structure

```
1  void parser(hls::stream<Packet> &qpkt,
2              byte *buff) {
3    bool accept = false; state = START;
4    Packet pkt;
5    switch(state) {...}  // Switch for layer 1
6    switch(state) {...}  // Switch for layer 2
7    ...            // Switches for middle layers
8    switch(state) {...}  // Switch for layer N
9    if (accept)
10       qpkt.write(pkt);
11 }
```

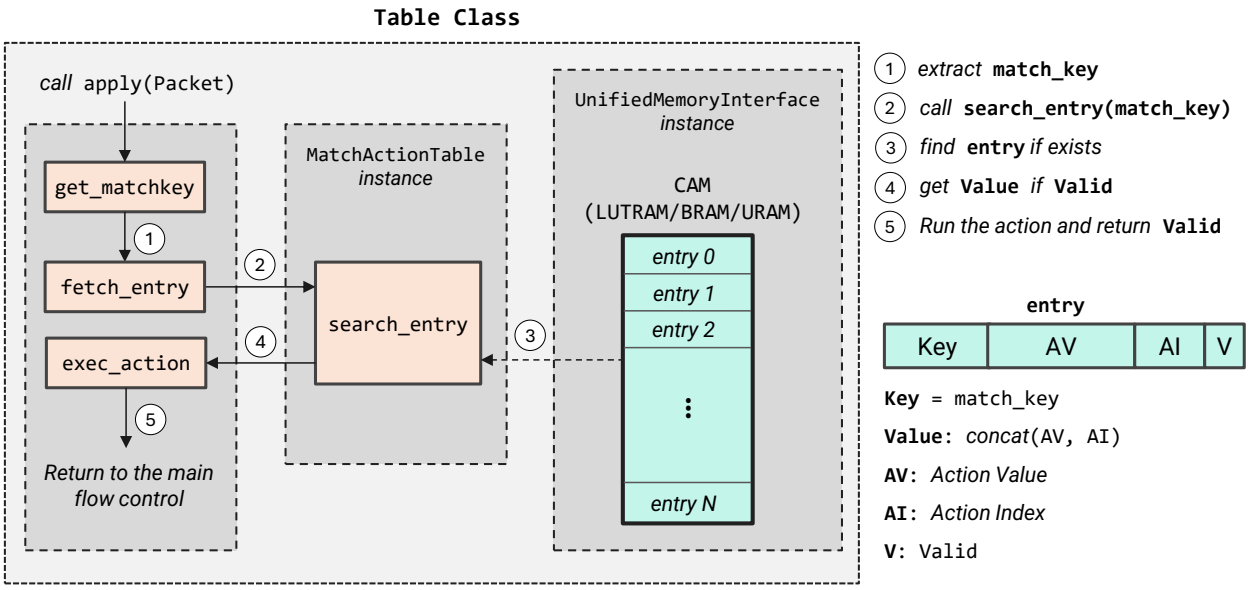

Figure 7.4 Steps to call the `apply()` in the table handler object

headers in the correct sequence. The deparser function iterates through each header, invoking the serialize function for each header type (e.g., Ethernet, IPv4) and appending it to the output buffer. By processing headers according to the defined packet structure, the deparser constructs a binary sequence ready for transmission, preserving the intended packet format. Listing 7.2 shows the internal structure of the generated deparser in C++. It serializes each header after checking its validity.

Match-Action Processing

The Match-Action Processing in Fig. 7.3 is configured according to the given P4 code to manipulate packets. This module comprises all objects related to the match-action tables

Listing 7.2 Deparser code structure

```
int deparser(byte *buff, Packet &pkt) {
  int count = 0;
  if (pkt.valid[H_ETHERNET]) {
    serialize(&buff[count], pkt.eth);
    count += H_ETHERNET_SIZE;
  }
  if (pkt.valid[H_IPV4]) {
    serialize(&buff[count], pkt.ipv4);
    count += H_IPV4_SIZE;
  }
  // Serialing other headers
  ...
  return count;
}
```

and determines the order of the calling table's methods.

Fig. 7.4 illustrates the structure of a generated templated table class. Each table class includes a handler and an accessor object to access the memory. The accessor object calls a specific function of the unified memory interface, which handles accessing the memory and retrieving the corresponding table entry according to the memory type (Section 7.3.4). The main process begins when its `apply` function is called. First, the `get_matchkey` function extracts relevant fields from the packet to form the match keys. These keys are then passed to the `fetch_entry` function, which calls the `search_entry` method within the `MatchActionTable` instance. Once this function identifies the matching entry, it retrieves its value to execute the specified action. Finally, control is returned to the main processing flow, performing the rest of the match-action processing for the packet. The primary template structure for each table class is illustrated in Listing 7.3. For example, the parameters `TS_ExampleTable`, `MW_ExampleTable`, `AW_ExampleTable`, and `MK_ExampleTable` correspond to the table size, match key width, action value width, and match kind, respectively, and are designated to the `ExampleTable` class.

CAM Implementation

In P4THLS, match-action tables are implemented using different types of Content-Addressable Memory (CAM) structures, depending on the kind of match operation required by the P4 program. Each match kind in P4 maps to a specific CAM type in the implementation:

- Exact Match is implemented using Binary CAM (BCAM), which performs a direct key-to-entry lookup.

Listing 7.3 Example of a Match-Action Table Implementation

```
typedef ap_uint<MW_ExampleTable> key_type;
typedef ap_uint<AW_ExampleTable> value_type;
class ExampleTable {
public:
  ExampleTable() {}
  // Extract the matching key and return it
  key_type get_matchkey(Packet &pkt) const {...}
  // Find the table entry and call actions
  bool apply(Packet &pkt) {
    const key_type key = get_matchkey(pkt);
    value_type value;
    bool found = impl.search_entry(key,value);
    if (found) {
 // Call an action based on the action index
 // otherwise, call the default action
    ...
    }
    return found; // return true if hits
  }
  // Instantiate the template class
  MatchActionTable<TS_ExampleTable,
                   MW_ExampleTable,
                   AW_ExampleTable,
                   MK_ExampleTable> impl;
};
```

- Longest Prefix Match (LPM) uses a Semi-Ternary CAM (STCAM) for prioritized lookups, ideal for IP routing.
- Ternary match supports wildcards ("don't care" bits) and is implemented with a Ternary CAM (TCAM).

These CAMs are constructed using the HLSCAM library [12], a templated C++ framework designed for HLS targeting FPGAs. The HLSCAM library provides multiple optimization modes: Brute-Force (BF) mode prioritizes design simplicity at the cost of increased lookup latency; Balanced (BL) mode offers a trade-off between area and speed; and High-Speed (HS) mode is optimized for low latency, high throughput packet processing. In the BL mode, the memory is divided into PF (Partition Factor) blocks, each holding PD (Partition Depth) entries [12]. This partitioning increases parallel memory access but grows distributed RAM occupation.

In this work, the appropriate CAM structures are selected automatically by the P4THLS based on the match type specified in the input P4 code. In addition, the BL mode is primarily used to meet performance goals. The optimization mode for each table can be

overwritten during the P4-to-HLS translation by setting command parameters or manually tweaking them after code generation through the generated configuration header file.

P4THLS contributes to HLSCAM by adding a Hash-Based (HB) BCAM to optimize exact match tables further. Instead of performing parallel comparisons across all entries, the HB BCAM uses a hash function to compute a direct index for table access, reducing both resource usage and lookup latency. Two hashing functions are supported: FNV1a [106], a lightweight, low-latency hash function suitable for latency-critical applications, and Jenkins [107], which offers better hash distribution and fewer collisions, though with slightly higher latency.

7.3.3 P4THLS Parameterization and Flexibility

One of the distinguishing features of P4THLS is its high degree of parameterization, enabling users to adapt the HLS-based design to diverse performance and resource constraints without extensive hardware expertise. As summarized in Table 7.2, configurable parameters include AXI bus width, internal queue depth, CAM architecture, and CAM memory type.

Increasing the bus width reduces the number of cycles required for parsing and deparsing, thereby improving throughput and lowering latency, but it also increases design complexity during place-and-route. Internal queue sizes can be tuned to improve tolerance to burst traffic and prevent packet loss, subject to the available on-chip memory budget. The CAM structure directly influences both latency and resource usage. For instance, a hash-based BCAM is generally simpler, faster, and more resource-efficient than a TCAM. The memory type used for CAMs is another critical design choice that affects the maximum table capacity. Selecting BRAM over LUTRAM or FFs, and URAM over BRAM, allows the implementation of larger tables, albeit at different resource trade-offs. By adjusting these parameters, P4THLS can be tailored to balance latency, throughput, and FPGA resource utilization according to specific deployment requirements.

7.3.4 Unified Memory Interface

FPGAs provide a variety of memory resources, including distributed memory (LUTRAM), Block RAM (BRAM), and Ultra RAM (URAM), each with unique architectural features. LUTRAM is implemented using FPGA Look-Up Tables (LUTs) within logic slices. BRAM typically offers larger capacity (up to 36 Kb per block), with flexible word widths (up to 72 bits) and dual read/write ports. URAM, with significantly higher capacity (up to 288 Kb per block), supports a word width of up to 72 bits and a single read/write port.

P4THLS introduces a unified memory interface seamlessly integrating various FPGA on-chip

Table 7.2 TP4THLS Parameters and Affected Measures

Parameter	**Affected Measures**
Bus Width	Throughput, latency
Internal Queue Size	Packet loss tolerablity, Memory occupation
CAM structure	Latency, Resource occupation
CAM memory type	Memory resource balance

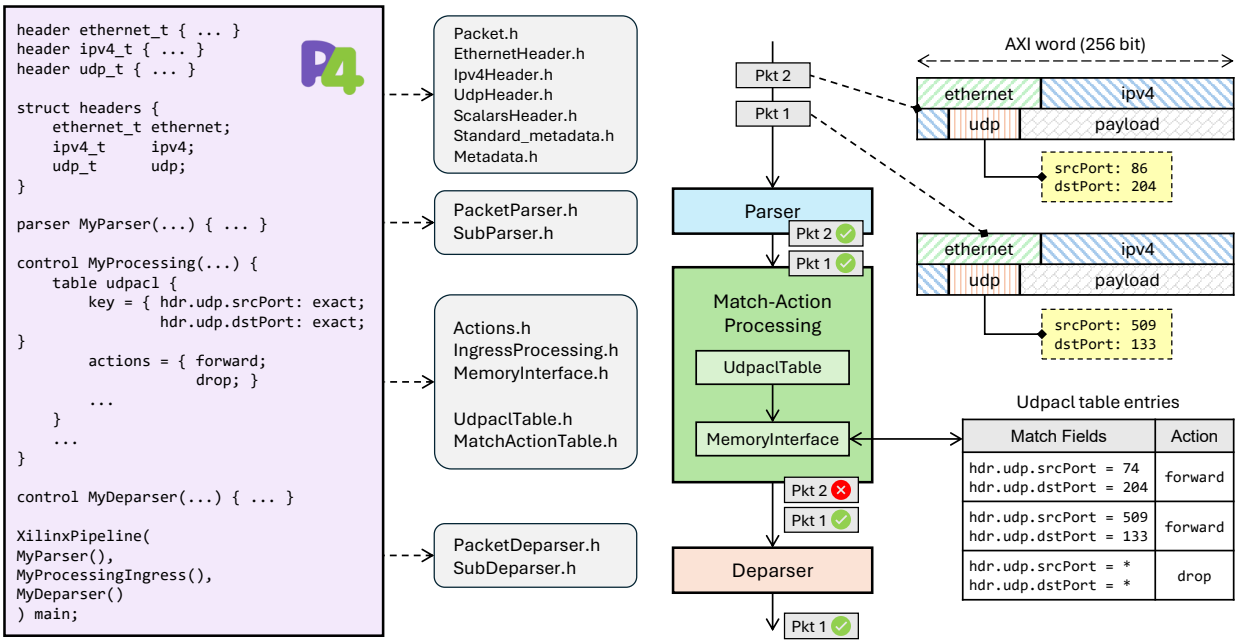

Figure 7.5 Example: An overview of the C++ generated kernel of a UDP Access Control List using the P4THLS

memory resources. This interface abstracts the underlying memory architecture, enabling each match-action table and CAM structure to be mapped to LUTRAM, BRAM, or URAM without modifying the table handler or CAM objects. The desired memory type for each table is specified through simple parameter adjustments. This flexibility simplifies memory balancing and improves scalability, allowing the packet processing engine to efficiently support complex P4 applications with numerous match-action tables.

The unified memory interface allows users to select the memory resource (BRAM, URAM, or LUTRAM) through a simple configuration. This is done by passing the memory type parameter, *mtype*, to the Python-based P4-to-HLS converter. The interface then automatically maps the match-action tables to the chosen memory type, eliminating the need for manual

editing of the generated HLS code and providing both flexibility and ease of use.

7.3.5 P4THLS Example: UDP Access Control List

To demonstrate the functionality of the P4THLS framework, we present a simple UDP Access Control List (ACL) example generated by the Code Generation Kernel, along with a sample packet structure comprising Ethernet, IPv4, and UDP headers, as shown in Figure 7.5. For each P4 program, P4THLS produces a set of core C++ files such as `PacketParser.h`, `PacketDeparser.h`, and `IngressProcessing.h` that are customized based on the input P4 code. In addition, the framework generates other source files dynamically, including header definitions and table-specific implementations such as `UdpAclTable.h`.

In this example, the `udpacl` table contains two rules that forward packets when both the source and destination UDP ports match specific values; all other packets are discarded. Among the incoming packets, Pkt 1 matches the first rule and will be forwarded, whereas Pkt 2 does not match any rule and will be dropped.

7.3.6 Dataflow and Performance Characteristics

The proposed architecture follows a three-stage streaming pipeline where the parser, match-action tables, and deparser are interconnected by FIFO buffers. Figure 7.6 illustrates the pipeline dataflow of the P4THLS kernel for a single input and output port from a latency point of view. Each stage processes packets independently, enabling the design to sustain high throughput. The parser comprises two sub-stages: Fetch, which reads incoming AXI-Stream words from the interface, and PG (Parser Graph), which reconstructs packets according to the P4-defined parse graph. Similarly, the deparser consists of SPF (Serialize Packet Fields), which serializes processed packet fields, and Transmit, which sends them back to the AXI-Stream output interface.

Eq (7.1) relates the pipeline latency to the latency of each stage, respectively, according to Figure 7.6:

$$\begin{aligned}\text{Pipeline Latency} = {} & L_{\text{Fetch}} + L_{\text{PG}} + \\ & L_{\text{FIFO1}} + L_{\text{Match-Action}} + L_{\text{FIFO2}} + \\ & L_{\text{SPF}} + L_{\text{Transmit}}\end{aligned} \tag{7.1}$$

Although the pipeline latency can be obtained by summing the latencies of individual stages, the throughput depends on a more complex set of parameters and runtime conditions. The

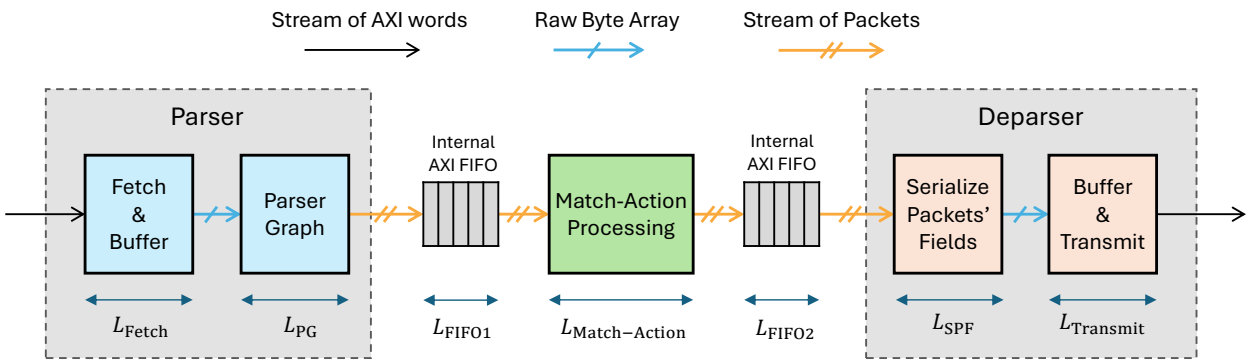

Figure 7.6 A latency-aware schematic of the P4THLS kernel's pipeline

following analysis examines the parameters that most strongly influence sustained throughput. The Fetch and Transmit stages operate at one AXI word per clock cycle, allowing them to sustain the maximum possible bandwidth and preventing them from becoming throughput bottlenecks. Under no backpressure conditions, each packet can be forwarded to the next stage in the following clock cycle, and the latency of intermediate FIFOs can be neglected. Consequently, the critical contributors to pipeline performance are PG, Match–Action, and SPF, as they determine both latency and sustained throughput. To maintain line-rate operation, each of these stages must process packets fast enough to avoid stalling the pipeline. For example, the PG stage should complete the packet reconstruction before the next packet is buffered; otherwise, incoming bytes will accumulate in the byte array, risking overflow. Similarly, the SPF stage should serialize and pack packets at least as fast as the transmit stage consumes them. The Match–Action stage should also adhere to this rule; if its processing delay exceeds the parser/deparser rate, it will become the bottleneck. To prevent stalls and consequently decrease the possibility of overflow and throughput degradation, pipelining is enabled within the stages PG, Match-Action, and SPF to prevent stalls. Although the PG and SPF blocks are typically mapped to forward-pipelined processes, the match-action logic can be more complex, especially when tables are accessed multiple times in different sequences, sometimes limiting the extent of internal pipelining depending on the capabilities of the HLS tool.

Eq. 7.2 indicates how achievable bandwidth (BW) can be evaluated based on operating clock frequency and bus width.

$$\text{BW (Gbps)} = \frac{\text{Clock Freq. (MHz)} \times \text{Bus Width (bit)}}{1000} \tag{7.2}$$

The maximum achievable bandwidth is directly proportional to the configured bus width when the operating frequency is fixed. For example, at 250 MHz, the bandwidth scales from 16 Gbps with a 64-bit interface to 128 Gbps with a 512-bit interface. Wider bus widths are particularly advantageous for high-throughput applications, but this comes at the cost of increased resource utilization and potential challenges in FPGA routing and timing closure.

7.4 Evaluation and Performance Analysis

This section evaluates the P4THLS framework, focusing on its performance across different architectural configurations and comparing it with existing alternatives. The goal is to assess the flexibility and effectiveness of P4THLS using performance metrics such as throughput, latency, and resource utilization. All results are obtained using AMD Vitis HLS 2023.2 for high-level synthesis and AMD Vivado 2023.2 for post-place-and-route analysis. The implementation targets the AMD Alveo U280 FPGA, a high-throughput, low-latency platform common in edge and data center environments.

7.4.1 Analysis of Different CAM Parameters

This evaluation investigates how different match-action table configurations affect the performance and resource utilization of the generated packet processing IP. Specifically, the focus is on comparing the impact of match kinds under varying table sizes, considering both logic consumption and pipeline intervals.

Table 7.3 summarizes the occupied resources and pipeline intervals for match-action tables implemented using the TCAM, STCAM, and BCAM structures. Partial memory partitioning is used for TCAM, STCAM, and BCAM implementations to parallelize comparisons across table entries, while a hash-based lookup mechanism is applied in the BCAM case.

TCAM requires 2 and 4 clock cycles to complete each lookup for table sizes of 128×32 and 256×64, respectively. For STCAM, these values increase to 3 and 5 cycles, adding an extra cycle compared to the ternary configuration. Additionally, due to the more complex prefix-matching logic, the LPM implementation uses more LUT and FF resources than its ternary counterpart. The conventional BCAM design consumes approximately half the LUT and FF resources of the TCAM implementation for both 128×32 and 256×64 table sizes, while still requiring 2 and 4 cycles per lookup, respectively.

However, using a hash-based BCAM significantly improves resource efficiency, despite a non-zero probability of hash collisions. This hashed version supports larger table sizes without increasing the lookup interval, thanks to a lightweight, low-latency hash function. It also

Table 7.3 Performance results for different match kinds used in match-action tables (BL optimization is used for TCAM, STCAM, and BCAM with PF = 32)

CAM Type	Match Kind	Memory Type	CAM Size (Depth×Width)	Pipeline Interval (Cycles)	Resource Utilization			
					LUT	FF	BRAM	DSP
TCAM	Ternary	LUTRAM	128×32	2	4,984	4,390	0	0
		LUTRAM	256×64	4	9,974	8,684	0	0
STCAM	LPM	LUTRAM	128×32	3	6,538	5,236	0	0
		LUTRAM	256×64	5	12,246	10,398	0	0
BCAM	Exact	LUTRAM	128×32	2	2,779	2,303	0	0
		LUTRAM	256×64	4	5,362	4,558	0	0
HB BCAM	Exact	LUTRAM	128×32	1	207	161	0	0
		LUTRAM	256×64	1	677	579	0	0
		LUTRAM	1024×64	1	1,723	613	0	16
		LUTRAM	8192×64	1	12,659	950	0	16
		BRAM	128×32	1	102	135	2	0
		BRAM	256×64	1	302	494	2	0
		BRAM	1024×64	1	199	518	3	16
		BRAM	8192×64	1	239	568	16	16

offers better memory allocation control, distributing storage across LUT, BRAM, or URAM based on the required capacity. The hash functions may consume up to 16 DSP slices for large CAMs to support higher arithmetic complexity. However, in smaller CAMs with fewer computation bits, modern HLS tools automatically infer LUT-based hash function implementations to avoid inefficient DSP usage. This selective resource mapping optimizes area utilization across different design scales while maintaining consistent performance.

7.4.2 Variable Bus Width

The AXI bus width directly affects the data transfer rates in the data plane switch. Figure 7.7 shows the latency of a UDP Firewall for different packet sizes across variable AXI bus widths. As expected, increasing the bus width reduces latency, with narrower widths (e.g. 64-bit) resulting in higher latency due to extra cycles needed to fetch and transmit packets. This effect is more pronounced for larger packets, such as 512 bytes, where latency drops significantly as the bus width increases from 64-bit to 512-bit.

The bus width directly affects L_{Fetch} and L_{Transmit}, while the latency of the other stages remains unchanged. Narrower buses reduce resource consumption because fewer parallel logic

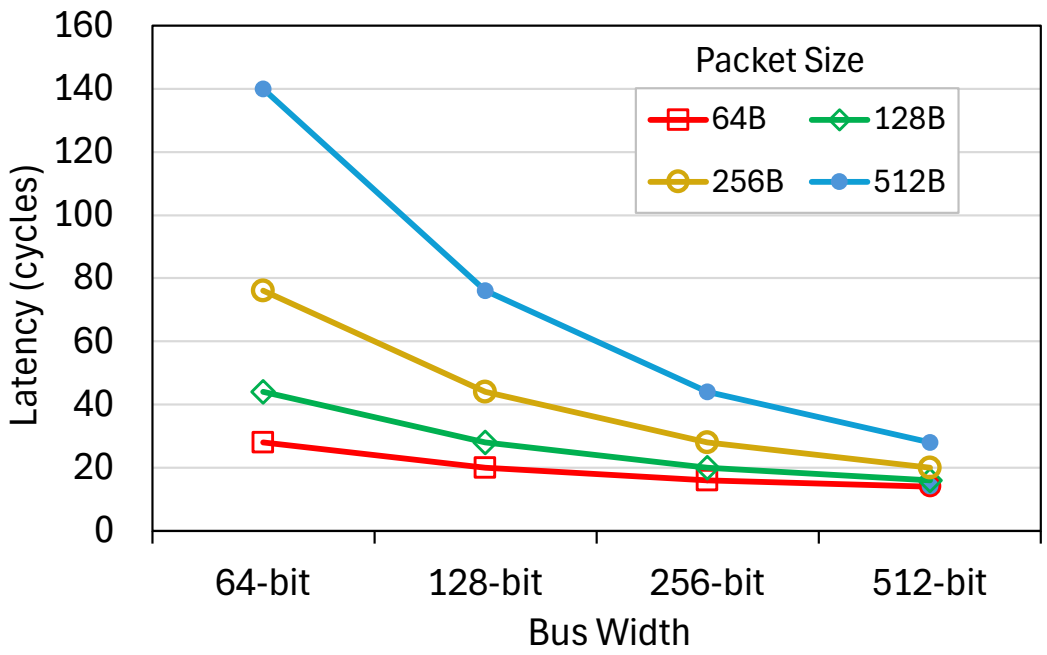

Figure 7.7 Impact of bus width on latency considering various packet sizes for a UDP Firewall application (f_{clk} = 250 MHz)

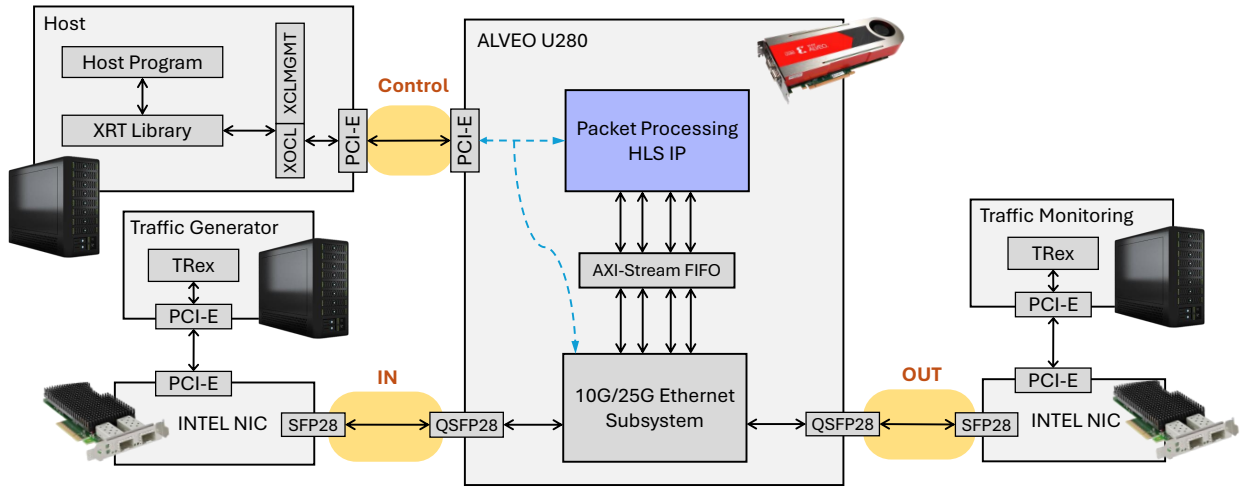

Figure 7.8 Experimental Setup - controller, FPGA block design, and traffic generation setup

elements are required, simplifying routing and easing timing closure. However, narrower buses increase the number of clock cycles needed to receive and transmit a packet, thus increasing latency. In contrast, wider buses allow packets to be processed in fewer cycles, reducing pipeline latency. Still, they also introduce more parallel data paths that increase

LUT and register usage and make the place-and-route process more complex. This trade-off is fundamental when tuning the design for specific performance or resource constraints. For example, for a 64-byte packet, four cycles are required for both the fetch and transmit stages, meaning L_{Fetch} and L_{Transmit} are both four cycles. However, with a 512-bit bus width, only one cycle is needed to fetch and transmit.

Table 7.4 Comparison of P4THLS, VitisNetP4, and P4-to-FPGA

Feature	**P4THLS** (this work)	**VitisNetP4** [57]	**P4-to-FPGA** [66]
Customized Bus-width	Yes	Yes	No
Full packet processing pipeline	Yes	Yes	Yes
Memory Interface Flexibility	Yes	Yes	No
Multiple In/Out Ports	Yes	No	No
Built-in Metadata	Limited	Yes	Limited
Extern and Stateful Objects	Limited	Yes	Undisclosed
Target Language Code	C++	Verilog	VHDL
Editable Generated Code	Yes	No (encrypted)	No
Full FPGA Implementation Flow	Yes	Partial	No
Project Status and Support	Open-source Framework	Commercial Tool	Closed-source

7.4.3 Qualitative Comparative Analysis

The comparative analysis highlights key strengths and limitations of the P4THLS framework compared to VitisNetP4 IP [57] and P4-to-FPGA [66]. Table 7.4 shows that P4THLS supports customizable bus widths, multiple input/output ports, and a flexible memory interface, providing strong adaptability across different FPGA architectures. It uses C++ as the target language and generates editable code, making it ideal for open development, research, and integration into custom workflows. In contrast, VitisNetP4 is a commercial IP with support for rich metadata and stateful externs, but it is closed source and encrypts the generated code, restricting customizability. While P4-to-FPGA is valuable from an academic standpoint, it lacks support for bus-width configuration, unified memory abstraction, and access

to its generated HDL code. Notably, only P4THLS offers a complete FPGA implementation flow, automating the entire process from P4 code analysis to generating a synthesizable IP core, creating the necessary control-plane interface, and packaging the module for direct integration into an FPGA block design using tools like AMD Vivado and Vitis. Although VitisNetP4 provides a prebuilt control-plane driver, it still necessitates additional IP cores, parameter tuning, and integration scripts for deployment on AMD UltraScale or newer FPGAs.

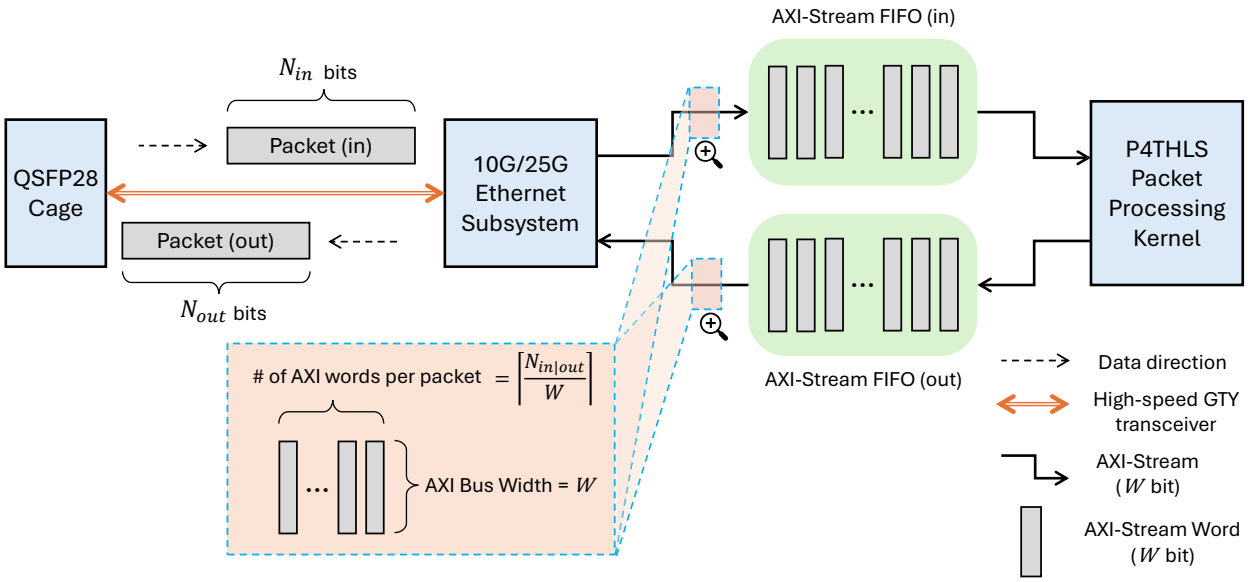

Figure 7.9 Packet flow between the QSFP28 cage and the P4THLS packet processing kernel

7.4.4 Empirical Comparative Analysis

Experimental Setup

Figure 7.8 illustrates the experimental setup used to evaluate the P4THLS framework in a realistic environment to validate functionality and applicability. It was also designed to measure the supported bandwidth and end-to-end latency to present comparative results. The experiments are conducted on an AMD Alveo U280 FPGA card, a data center–class platform optimized for high-throughput and low-latency application acceleration. Traffic is generated and received using two Intel XXV710 Ethernet Network Adapter interfaces, each capable of 25 Gbps full-duplex throughput. These interfaces are connected directly to the FPGA high-speed transceivers to provide stress testing of the data plane under line-rate conditions. The TRex Traffic Generator [74] produces synthetic traffic patterns and captures

processed packets for analysis.

Within the FPGA, the 10G/25G Ethernet Subsystem IP core converts physical Ethernet traffic from the QSFP28 cages through high-speed GTY transceivers into AXI-Stream format, the standard interface for high-bandwidth data transfer across AMD FPGA fabrics, as shown in Figure 7.9. Incoming packets are segmented into multiple AXI-Stream words, which are first buffered in AXI-Stream FIFOs to ensure smooth delivery to the packet processing kernel. For instance, for an incoming packet of size N_{in} bits and an AXI bus width of W bits, the packet is segmented into $ceil(N_{in}/W)$ AXI-Stream words. The parser then reassembles these words to reconstruct the original packet. After processing, the reverse operation is performed to serialize the packet back into AXI-Stream words for transmission. The design also supports multi-lane streaming, enabling four bidirectional 25 Gbps lanes per QSFP28 interface. Incoming traffic across these lanes is distributed using a round-robin scheduler to balance processing across the data path.

A host machine, connected to the FPGA via PCIe, provides control-plane functionality. It manages the 10G/25G Ethernet subsystem and monitors the status of the instantiated IP blocks. The host is also responsible for running the host program to control tasks such as configuring match-action tables, reading and writing to internal registers and counters, and triggering run-time reconfiguration or debugging operations.

Performance Metrics Definition

To evaluate the proposed packet processing kernel, metrics should be defined. Throughput (TP) is defined as the rate of bits processed per second, which is measured in gigabits per second (Gbps) and calculated using Eq. 7.3:

$$Throughput\ (TP) = f_{max} \times W_{AXI} \tag{7.3}$$

where the f_{max} is the maximum frequency and W_{AXI} is the AXI bus width. Since other work in the literature may utilize different FPGA devices, a normalized throughput ($N.\ TP$) metric is introduced to allow for a fair comparison. It is defined in Eq. 7.4:

$$Normalized\ \ TP = Scaling\ Factor \times TP \tag{7.4}$$

where $Scaling\ Factor$ is a coefficient that compensates for differences in FPGA technology speed, making performance metrics more comparable across different devices. The

Scaling Factor is computed using Eq. 7.5 [12]:

$$Scaling\ Factor = \frac{Technology\ (nm)}{28\ (nm)} \times \frac{1.0}{VDD} \tag{7.5}$$

where *Technology* refers to the FPGA CMOS technology size and the VDD refers to the corresponding supply voltage.

Table 7.5 Different FPGA parameters.

FPGA Family	CMOS Technology	VDD	Scaling Factor
Virtex-7, Kintex-7	28 nm	1.0 V	1.0
Virtex Ultrascale+	16 nm	0.85 V	0.67

The Virtex-7 family is used as the baseline FPGA, as it is fabricated with a 28 nm process and operates at 1.0 V supply voltage. Table 7.5 lists the related FPGA parameters used to calculate normalized throughput. It provides the relevant FPGA parameters, including those for Kintex-7, Virtex-7, and Virtex Ultrascale+, enabling the calculation of scaling factors for each.

P4-to-FPGA Applications

Table 7.6 compares the utilization, latency, throughput, and normalized throughput of P4THLS and VitisNetP4 resources with the results presented in P4-to-FPGA [66] in three representative P4 applications. The base clock rate for P4THLS and VitisNetP4 was set to 250 MHz. Hence, the actual throughput is calculated as the product of the base clock frequency and the AXI bus width; whereas the maximum throughput represents the theoretical upper bound, based on each design's maximum post-place and route clock frequency. The normalized throughput also reflects the maximum throughput after applying the technology scaling factor.

Module latency refers to the internal delay introduced by the packet processing IPs. In contrast, end-to-end latency captures the total delay experienced from packet transmission at the source to reception at the destination, including peripheral and I/O delays. The selected applications, including Basic Switch, MRI (Multi-hop Route Inspection), and Calc, correspond to case studies evaluated in the P4-to-FPGA work, providing a fair comparison, as that framework lacks publicly available tooling to reproduce results for arbitrary P4 programs. Basic Switch performs standard L3 forwarding; MRI incorporates scaled-down

Table 7.6 Comparison of resource utilization, latency, and throughput between P4THLS and P4-to-FPGA IP for three P4 Applications (f_{clk} = 250 MHz)

P4 Application	Work	Bus Width (bit)	Resources			Latency		Max. Clock (MHz)	Actual TP (Gbps)	Max. TP (Gbps)	N.TP (Gbps)
			LUT	Registers	BRAM	Module (cycles)	End-to-end(us) /std.dev(σ)				
Basic Switch	P4THLS (this work)	512	6629	4209	4	10	8.1/±1.34	262.2	128	134.2	90.2
	P4THLS (this work)	256	5971	3584	4	12	8.1/±1.38	271.3	64	69.5	46.7
	VitisNetP4 [57]	256	5211	8997	9	46	8.1/±1.46	259.1	64	66.3	44.6
	P4-to-FPGA [66]	272	5636	1943	2	7	NA	176.9	NA	48.1	48.1
MRI	P4THLS (this work)	512	11046	10254	4	15	8.1/±1.32	255.4	128	130.8	87.9
	P4THLS (this work)	256	9721	9156	4	17	8.1/±1.28	261.7	64	67.0	45.0
	VitisNetP4 [57]	256	9557	12457	9	48	8.2/±1.36	257.0	64	65.8	44.2
	P4-to-FPGA [66]	880	8003	8354	2	10	NA	168.3	NA	148.1	148.1
Calc	P4THLS (this work)	512	1471	2619	1	6	8.1/±1.41	279.8	128	143.3	96.3
	P4THLS (this work)	256	864	2029	1	8	8.0/±1.4	298.7	64	76.5	51.4
	VitisNetP4 [57]	256	2236	4299	1	27	8.1/±1.33	271.8	64	69.6	46.8
	P4-to-FPGA [66]	240	693	1445	0	4	NA	307.5	NA	73.8	73.8

in-band network telemetry; Calc represents a minimal form of in-network computation. All comparisons focus solely on essential packet headers. A bus width of 256 bits is selected as the baseline for evaluation; however, a 512-bit bus width is also used in our work to enable a more direct comparison with P4-to-FPGA, which reports a distinct bus width for each application. In addition, the table sizes are set to 256 entries for all methods to standardize the comparison.

In the Basic Switch application, P4THLS with a 512-bit bus width achieves the highest normalized throughput of 90.2 Gbps, delivering almost double the normalized rate of VitisNetP4 (44.6 Gbps) and significantly higher than P4-to-FPGA (48.1 Gbps). Even with the baseline

256-bit bus, P4THLS sustains 46.7 Gbps normalized throughput while maintaining a low end-to-end latency of 8.1 µs and using fewer registers than VitisNetP4. Although P4-to-FPGA exhibits a lower module latency, its maximum throughput remains lower. In terms of module latency, P4THLS requires only 10–12 cycles compared to 46 cycles for VitisNetP4, indicating a more efficient packet processing pipeline. However, P4-to-FPGA shows the lowest module latency at 7 cycles. For the MRI application, P4THLS with 512 bits reaches a normalized throughput of 87.9 Gbps, nearly doubling the rates of VitisNetP4 with 44.2 Gbps. While P4-to-FPGA reports the highest throughput of 148.1 Gbps, this is achieved using an unusually wide 880-bit bus, which increases design complexity, reduces operating frequency, and is less practical for typical FPGA deployments. The module latency for P4THLS remains in the range of 15–17 cycles, significantly lower than VitisNetP4's 48 cycles, while P4-to-FPGA achieves 10 cycles. In the Calc application, which involves lightweight arithmetic processing, P4THLS with a 512-bit bus width attains the highest normalized throughput across all experiments, reaching 96.3 Gbps with an end-to-end delay of around 8.0 µs. The 256-bit version still outperforms VitisNetP4 in normalized throughput while consuming substantially fewer logic resources. Module latency for P4THLS is only 6–8 cycles, considerably lower than VitisNetP4's 27 cycles, though slightly higher than P4-to-FPGA's 4 cycles.

As shown in Table 7.6, the proposed P4THLS consistently consumes slightly more LUTs and registers than P4-to-FPGA in all scenarios. This is primarily due to differences in bus width configuration. One reason is that the P4-to-FPGA uses a specific bus width, the same length as the headers, which allows extra pipeline stages to buffer packet chunks before parsing and deparsing. The fewer cycle stages confirm this analysis, indicating only one clock cycle for the parser and one for the deparser. This assumption is not always realistic, since changing the bus width at runtime is impossible based on the packet size. Additionally, wider buses may complicate the place-and-route phase of FPGA implementation. This limitation becomes apparent in the MRI application, which requires a bus width of at least 880 bits to match the header size. Although this enables a higher throughput than P4THLS, it significantly reduces the maximum clock frequency to 168.3 MHz. In contrast, P4THLS and VitisNetP4 use a lower 256-bit AXI stream bus, requiring more cycles to process wide headers like MRI, but allowing for a much higher clock rate and a more simplified general implementation.

Another factor contributing to P4THLS's slightly higher resource usage and latency than P4-to-FPGA is its reliance on high-level synthesis. P4-to-FPGA employs HDLs for direct mapping to hardware, which typically yields more resource-efficient results. In contrast, P4THLS uses a templated C++ HLS approach, which provides better flexibility for customization and integration at the cost of some additional logic overhead. Importantly, P4THLS is released as an open-source toolchain, enabling automatic regeneration of C++ HLS modules from

input P4 programs. In contrast, the internal structure of the P4-to-FPGA solution is not publicly disclosed.

The similarity in end-to-end latency between P4THLS and VitisNetP4 can be attributed to their comparable experimental setup and peripheral components. Both frameworks are deployed on the same FPGA platform and tested using identical Ethernet interfaces, PCIe communication channels, and traffic generation tools. As a result, the overall latency is dominated mainly by non-processing factors such as transmission delays across physical links, propagation delays, and queueing within the I/O subsystems. These external delays overshadow the internal processing time of the packet processing modules, causing the minor differences in pipeline latency between the two implementations to be effectively masked in the total end-to-end measurement.

As a commercial solution, VitisNetP4 does not expose internal configuration options such as packet size constraints for fine-grained control over synthesis and latency measurements. Consequently, the reported performance and resource utilization figures are derived from estimates of the synthesis tool rather than accurate cycle simulations or real measurements. They may be overestimated compared to our testable implementations. Furthermore, VitisNetP4 IP includes industrial-grade features, such as backpressure detection and advanced flow control mechanisms, which introduce additional logic overhead and increase the required processing cycles. These enhancements are valuable for production-grade robustness, but come at the cost of increased resource consumption and latency.

Large-Scale Table Size Applications

Table 7.7 presents a comparative analysis between the proposed method and VitisNetP4 for large-scale table sizes and varying AXI bus widths, using two representative applications: packet classification and UDP firewall. The packet classification application inspects five header fields to identify and classify traffic flows, while the UDP firewall filters packets based on UDP source/destination IP addresses and ports. Two configurations are considered for each application: one with 1K table entries and a 64-bit AXI bus, and another with 64K table entries and a 256-bit bus. In both cases, BRAMs are employed for the smaller configuration, and URAMs are used for the larger one, leveraging the configurable memory interface supported by both architectures. The evaluation assumes a fixed packet size of 1024 bits, comprising Ethernet, IPv4, UDP headers, and a sufficient payload. Since the end-to-end delays of P4THLS and VitisNetP4 are consistently similar across all evaluated applications, as reported in Table 7.6, they are omitted here to avoid redundancy, as they do not provide additional information for comparison.

Table 7.7 Comparison of resource utilization and latency between P4THLS and AMD Xilinx VitisNetP4 IP for two P4 Applications with varying parameters (f_{clk} = 250 MHz)

		Parameters	Design Parameters		Resource Utilization				
P4 Application	Work	Table Entries	BusWidth (bit)	Memory Type	LUT	Registers	BRAM	URAM	Latency (cycles)
Packet Classification	P4THLS (this work)	1K	64	BRAM	9,157	13,652	13	0	46
	VitisNetP4 [57]	1K	64	BRAM	5,550	8,799	12	0	55
	P4THLS (this work)	64K	256	URAM	9,622	15,451	0	64	28
	VitisNetP4 [57]	64K	256	URAM	8,179	13,902	0	60	48
UDP Firewall	P4THLS (this work)	1K	64	BRAM	9,539	9,085	13	0	41
	VitisNetP4 [57]	1K	64	BRAM	8,740	13,876	13	0	48
	P4THLS (this work)	64K	256	URAM	11,018	10,330	0	96	23
	VitisNetP4 [57]	64K	256	URAM	11,539	17,183	0	60	55

In the Packet Classification application, P4THLS consistently demonstrates a latency advantage over VitisNetP4, achieving 46 clock cycles at 1K table entries and 28 cycles at 64K entries, compared to 55 and 48 cycles for VitisNetP4. This reduction becomes more pronounced in the larger configuration. However, the latency improvement comes with higher logic utilization, as P4THLS consumes approximately 9.6K LUTs and 15.5K registers at 64K entries, while VitisNetP4 uses around 8.2K LUTs and 13.9K registers. The increased usage of resources in P4THLS reflects the overhead of managing wider match fields and deeper pipelining. Memory utilization (BRAM and URAM) remains comparable between the two solutions.

In the UDP Firewall application, where the match logic is narrower, P4THLS achieves both better latency and more efficient logic usage. It records 41 and 23 cycles at 1K and 64K entries, respectively, while VitisNetP4 requires 48 and 55 cycles for the same configurations. Furthermore, P4THLS consumes fewer LUTs and significantly fewer registers, about 10.3K

versus 17.2K, in the 64K-entry case. However, P4THLS uses more URAM blocks, 96 compared to 60, indicating a higher on-chip memory demand. This discrepancy stems primarily from VitisNetP4's internal memory management strategy, which more effectively consolidates unused space within URAM arrays allocated for match action tables. It is worth mentioning that the synthesis tool sometimes allocates a single BRAM for internal queueing, which is not used for storing table contents. This occurred in both BRAM-based versions of our work. In other words, 12 out of the 13 BRAMs used in both applications are dedicated to table storage.

Table entries in the evaluated designs often exceed the native width of on-chip memory blocks, which is 36 bits for BRAM and 72 bits for URAM in true dual-port mode. Consequently, multiple memory blocks must be cascaded to form a wider unified memory capable of storing complete table entries. This cascaded configuration can lead to under-utilizing allocated memory capacity, as not all bits within each block are necessarily used. Furthermore, HLS synthesis tools attempt to optimize memory allocation; however, the mapping strategy is strongly influenced by entry structure and field widths. For example, each table entry is approximately 120 bits wide in the packet classification design. Theoretically, this requires a minimum of four BRAMs to store each entry. In practice, more blocks are allocated because entries are split into multiple match fields of varying widths, and each field is often mapped to a dedicated set of memory blocks. This approach simplifies address generation, scheduling, and pipelining in the synthesized hardware at the expense of memory efficiency.

When table sizes scale, the number of memory blocks generally grows proportionally but not perfectly linearly due to the aforementioned partitioning and mapping considerations. For instance, scaling from 1K to 64K entries suggests an approximate 64× increase in memory requirements; however, in the 64K-entry configuration, URAM is employed instead of BRAM due to its much larger capacity (288 Kb vs. 36 Kb). Based on this substitution, 12 BRAMs at 1K scale correspond to roughly 96 URAMs at 64K scale. This scaling behavior matches exactly for the UDP firewall implementation, but slightly overestimates the actual allocation in packet classification because the synthesis tool optimizes packing when field alignment permits.

The latency behavior is largely determined by packet length and bus width. Considering 1024-bit packets and a 64-bit AXI data bus, 16 cycles are required to fetch the packet into the processing kernel and another 16 cycles to transmit the processed packet, resulting in 32 cycles dedicated solely to I/O transfer. In the packet classification application, this constitutes 32 of the total 46 pipeline cycles for both P4THLS and VitisNetP4. Increasing the bus width to 256 bits reduces the fetch and transmit stages to only 4 cycles each, theoretically

reducing total latency by 24 cycles. In practice, the improvement is slightly less because larger table sizes impose additional lookup cycles, which partially offset the gains from faster I/O. The reported latency for VitisNetP4 may also be inflated by its synthesis tool's modeling of variable packet lengths and other protocol-level factors, which are not part of the fixed-length measurement used in this work.

In general, resource usage trends across both applications confirm that the packet classification task, which involves larger and more complex tables, results in higher logic and memory demands for P4THLS. In contrast, for narrower tables like those in the UDP firewall, P4THLS exhibits more efficient utilization of logic resources compared to the commercial IP.

7.4.5 Discussion

The evaluation results highlight the effectiveness of P4THLS in delivering high-performance, low-latency packet processing for FPGA-based data plane applications. Its scalability is demonstrated through consistent performance across varying bus widths and table sizes, maintaining low latency and high throughput even under demanding configurations. The unified and configurable memory interface further reinforces this adaptability, dynamically balancing BRAM and URAM usage based on the application's resource and performance requirements.

As an open-source and highly customizable framework, P4THLS is a practical and accessible solution for researchers and developers targeting programmable data planes. One of its key strengths lies in providing a template-based C++ HLS project from P4 code, which users can readily modify and extend to accommodate diverse application needs. Thus, the integration of P4-defined behaviors into FPGA implementations is streamlined. P4THLS also includes a suite of high-speed, efficient CAM implementations, enabling the construction of match-action tables with customizable depth and width. Additionally, it supports various AXI bus widths, allowing users to trade off between processing latency and hardware complexity. Wider buses reduce the number of cycles required for parsing and deparsing, leading to lower latency, while narrower buses help simplify routing and reduce resource usage in more constrained designs.

The comparative analysis against VitisNetP4 and P4-to-FPGA confirms that P4THLS achieves competitive resource utilization while offering greater design flexibility. Its tunable architecture empowers engineers to optimize packet processing pipelines for various use cases, making it a robust and versatile platform for FPGA-based network processing.

7.5 Future Work

While the proposed P4THLS framework demonstrates flexibility and efficiency in transforming P4 programs into FPGA-ready HLS designs, several areas remain open for further enhancement. One primary direction addresses the limited support for built-in metadata, stateful objects, and custom externs. Expanding these capabilities would allow the framework to support more advanced P4 features and enable the implementation of richer network behaviors such as per-flow state tracking, queue management, and time-based control logic.

Another promising direction is integrating In-Band Network Telemetry (INT) support into the packet processing pipeline. INT enables real-time visibility into the network by embedding telemetry data directly within data packets.

7.6 Conclusion

This paper introduced P4THLS, a templated HLS framework that bridges high-level P4 programming with FPGA-based packet processing, enabling efficient and flexible deployment of data plane applications. Unlike existing solutions, P4THLS provides a fully automated workflow, from extracting P4 semantics to generating HLS-based IP cores, integrating control-plane drivers, and supporting out-of-the-box FPGA implementation using the AMD's Vitis toolchain.

Several innovative features distinguish P4THLS from existing solutions. A key contribution is the unified memory interface, which abstracts underlying memory architectures and allows seamless switching between LUTRAM, BRAM, and URAM without modifying tables or CAM logic. This enables users to balance memory usage and optimize for placement and routing in resource-constrained designs. Additionally, the framework supports customizable AXI bus widths, multi-port support, and a library of CAM structures, including hash-based and pipelined implementations for Exact, LPM, and Ternary matches, allowing fine-grained performance tuning based on application needs.

P4THLS also simplifies customization and encourages modularity, making it a practical tool for research and prototyping. Unlike commercial tools like VitisNetP4, which rely on encrypted output and limited configurability, P4THLS is fully open-source and editable, supporting advanced use cases and integration scenarios. Through extensive evaluation, P4THLS demonstrated competitive performance and scalability. It achieved throughputs of up to 76.5 Gbps using a 256-bit bus, and can exceed this throughput by increasing the bus width—albeit at the cost of increased place-and-route complexity. The architecture also showed efficient support for large-scale match-action tables, handling up to 64K entries and beyond with

competitive resource usage and lower latency than VitisNetP4. Across various case studies, P4THLS maintained end-to-end latencies around 8 µs, while delivering resource usage and bandwidth efficiency comparable or superior to existing tools like VitisNetP4 and P4-to-FPGA.

CHAPTER 8 ARTICLE 4: A STATEFUL EXTENSION TO P4THLS FOR ADVANCED TELEMETRY AND FLOW CONTROL

M. Abbasmollaei, T. Ould-Bachir and Y. Savaria, (submitted in 2025-10-14). "A stateful extension to P4THLS for advanced telemetry and flow control," submitted to *Future Internet*.

Abstract

Programmable data planes are increasingly essential for enabling in-band network telemetry (INT), fine-grained monitoring, and congestion-aware packet processing. Although the P4 language provides a high-level abstraction to describe such behaviors, implementing them efficiently on FPGA-based platforms remains challenging due to hardware constraints and limited compiler support. Building on P4THLS framework, which leverages HLS for FPGA data-plane programmability, this paper extends the approach by introducing support for P4-style stateful objects and a structured metadata propagation mechanism throughout the processing pipeline. These extensions enrich pipeline logic with real-time context and flow-level state, thereby facilitating advanced applications while preserving programmability. The generated codebase remains extensible and customizable, allowing developers to adapt the design to various scenarios. We implement two representative use cases to demonstrate the effectiveness of the approach: an INT-enabled forwarding engine that embeds hop-by-hop telemetry into packets and a congestion-aware switch that dynamically adapts to queue conditions. Evaluation of an AMD Alveo U280 FPGA implementation reveals that the proposed meter maintains rate measurement errors below 3% at 700 Mbps and achieves up to a 5× reduction in LUT and 2× reduction in Flip-Flop usage compared to existing FPGA-based stateful designs, substantially expanding the applicability of P4THLS for complex and performance-critical network functions.

Keywords
FPGA, High-Level Synthesis, P4THLS, Data Plane Processing, P4 language, Metadata, Statful Objects, In-band Network Telemetry, Flow Control

8.1 Introduction

Modern network infrastructures are increasingly dynamic and heterogeneous, supporting diverse applications that demand both high performance and adaptability [108]. Software-defined networking (SDN) has enabled programmability in the control plane; however, the

data plane has remained largely constrained to simple match–action processing [8]. This limitation restricts the deployment of advanced network functions that require persistent state and real-time metadata propagation. This gap is particularly evident with the rise of in-band network telemetry (INT) [81], congestion-aware forwarding [109], and fine-grained quality of service (QoS) enforcement [110, 111], which have further intensified the need for data plane architectures that support both high-performance packet processing and rich contextual state management.

Field-programmable gate arrays (FPGAs) have emerged as a compelling platform to implement programmable data planes due to their ability to achieve line-rate performance while maintaining reconfigurability [112, 113]. Unlike fixed-function application-specific integrated circuits (ASICs) or software-based solutions, FPGAs offer the unique combination of deterministic low-latency processing, massive parallelism, and runtime adaptability essential for next-generation network applications [11]. However, translating high-level data plane programs into efficient FPGA implementations remains challenging, particularly when applications require stateful processing and the propagation of structured metadata throughout the packet processing pipeline [114].

The P4 programming language has established itself as the *de facto* standard for describing programmable data plane behavior, providing hardware-agnostic abstractions for packet parsing, match-action processing, and packet modification [10]. While P4 compilers for switch ASICs have matured significantly, FPGA-targeted P4 implementations face fundamental limitations, particularly in supporting stateful objects such as registers, counters, and meters, as well as structured metadata fields that carry contextual information across pipeline stages [66].

In [4], we introduced P4THLS, a templated high-level synthesis (HLS) framework that automates the mapping of P4 data-plane applications to FPGAs, offering an open-source flow with configurable bus widths, unified memory abstraction, and optimized match-action table implementations. Although this framework demonstrated competitive throughput and latency compared to existing FPGA solutions, its support for advanced P4 constructs remained limited, particularly concerning stateful objects and structured metadata. This paper extends this line of work by incorporating native support for registers, counters, and meters, together with a structured mechanism for metadata propagation across pipeline stages[1].

The main contributions of this paper are summarized as follows:

1. Development of HLS-based implementations for P4 stateful objects, including registers,

[1] All implementation codes will be made publicly available in the P4THLS repository at `https://github.com/abbasmollaei/P4THLS` upon publication of this paper.

counters, and meters that provide persistent state management, supporting both direct (per table entry) and indirect operation modes, with line-rate performance.

2. Implementation of structured metadata fields, including ingress and egress timestamps, parsed byte counters, and custom metadata that propagate contextual information throughout the FPGA processing pipeline.

3. Seamless extraction of stateful objects, alongside other P4 constructs, from the native P4 code and their automatic translation into a templated HLS architecture.

4. Comprehensive evaluation through two representative use cases that demonstrate how a complex stateful application can be executed on an FPGA while maintaining line rate performance, that is, i) An INT-enabled forwarding engine; and ii) Flow control.

The remainder of this paper is organized as follows. Section 8.2 reviews background and related work on P4 programmability, stateful processing, and FPGA-based data plane design. Section 8.3 presents the proposed architecture, describing the extensions to P4THLS with structured metadata and native support for stateful objects. Section 8.4 reports experimental results on an AMD Alveo U280 platform, demonstrating the performance and resource efficiency of the proposed approach through representative use cases. Finally, Section 8.5 concludes the paper and outlines future research directions.

8.2 Background and Related Work

8.2.1 P4 Language and Data Plane Packet Processing

P4 is a domain-specific declarative language developed to program the data plane of networking hardware. First proposed in 2014 [10] and later extended in 2016 [104], it has evolved into two main versions: $P4_{14}$ and $P4_{16}$. The language was designed with three main objectives in mind: reconfigurability, independence from specific protocols, and portability across different targets. Thanks to these properties, P4 can be deployed on various hardware platforms without requiring detailed knowledge of their internal architecture [115].

The adaptability of P4 has driven its adoption in programmable switching devices based on the protocol-independent switch architecture (PISA) [10]. A PISA pipeline typically consists of three stages: a parser that extracts relevant packet fields, a sequence of match–action tables that apply rules to process the packets, and a deparser that reconstructs the modified packets for transmission. More broadly, this evolution reflects the rise of in-network computing,

where programmable data planes are increasingly leveraged for distributed processing and application offloading beyond simple forwarding tasks [116].

8.2.2 Stateful Processing in P4 Data Planes

The original P4 design emphasized protocol independence and pipeline reconfigurability but provided only limited persistent state through registers, counters, and meters. Several language extensions and frameworks were proposed to enrich state management in the data plane to address this. Early work investigated scalable monitoring for traffic statistics [117] and in-network key–value caching [118]. FlowBlaze.p4 introduced a structured mapping of Extended Finite State Machines (EFSMs) [119], while subsequent efforts proposed broader stateful extensions [120, 121]. More recently, EFSMs have been integrated as a native P4 construct, allowing the expression of complex behaviors [122].

Parallel interest has emerged in implementing stateful processing on programmable hardware. FlowBlaze was first deployed on an FPGA as a structured EFSM framework [123], inspiring hybrid systems that combine ASIC switching with FPGA co-processors for advanced telemetry [124]. Beyond EFSMs, stateful abstractions have been applied to security and machine learning (ML). For example, P4-NIDS leverages P4 registers and flow-level state for in-band intrusion detection [125], while MAP4 demonstrates how ML classifiers can be mapped to programmable pipelines [126]. Other heterogeneous designs combine programmable switches with FPGAs to realize stateful services at scale. For example, Tiara achieves terabit-scale throughput and supports tens of millions of concurrent flows by partitioning memory- and throughput-intensive tasks between FPGAs and PISA pipelines [29].

Recent efforts have also targeted stateful processing in FPGA-based P4 data planes. Building on the P4THLS framework [4], which translates P4 programs into synthesizable C++ for high-level synthesis, subsequent work has demonstrated the feasibility of compiling more complex P4 constructs to FPGA targets while maintaining programmability. One line of research demonstrates FPGA-based SmartNICs programmed with P4 and HLS to sustain near line-rate traffic analysis, showing that P4 abstractions can be combined with stateful extensions to support large-scale monitoring and telemetry [127]. Unlike these works, our contribution extends the P4THLS framework itself to natively support P4-style stateful objects and structured metadata propagation.

8.2.3 Meter Algorithms

Two widely adopted traffic metering algorithms are standardized in RFC 2697 [128] and RFC 2698 [129]. RFC 2697 defines the Single-Rate Three-Color Marker (srTCM), which regulates traffic using a single rate parameter, the Committed Information Rate (CIR), along with a Committed Burst Size (CBS) and Excess Burst Size (EBS). The packets are classified as green, yellow, or red based on token availability in two CIR-driven buckets. Owing to its simplicity, srTCM distinguishes between conformant and non-conformant traffic but does not capture peak-rate dynamics. RFC 2698 extends this model with the Two-Rate Three-Color Marker (trTCM), introducing both a CIR and a Peak Information Rate (PIR). This dual-rate design enables more fine-grained traffic characterization by enforcing both peak and long-term rates.

Building on these standards, recent work has proposed adaptive and predictive token-bucket variants. The dynamic priority token bucket tolerates temporary violations by lowering application priority rather than immediately falling back to best-effort service, thereby improving robustness against bursty traffic [130]. The prediction-based fair token bucket coordinates distributed rate limiters, improving fairness and bandwidth utilization in cloud environments [131]. In addition, TCP-friendly meters have been implemented in P4 data planes, addressing the limitations of commercial switch meters and significantly improving TCP performance [5].

Modern implementations must also consider hardware-specific constraints. TOCC, a proactive token-based congestion control protocol deployed on SmartNICs, addresses offload, state management, and packet-processing limitations while achieving low latency and reduced buffer occupancy compared to reactive schemes [132]. In programmable switches, active queue management based on traffic rates is completely implemented in PISA-style data planes, utilizing sliding windows of queue time to achieve fine-grained rate control within the constraints of match–action pipelines [133].

8.2.4 In-band Network Telemetry

In-band network telemetry (INT) embeds the device and path state directly into the packets, enabling real-time, hop-by-hop visibility without relying on external probes. Hence, network measurement has changed from control plane polling to data plane telemetry [134], with a key challenge being overhead. INT headers increase in size with path length, which affects latency-sensitive flows.

Recent work has shown that applying signal sampling and recovery to reconstruct missing

data from partial INT records can cut overhead by up to 90% [135]. A proposed approach combines INT with segment routing to reuse header space and maintain a constant packet size [136]. At the same time, another approach introduced a balanced INT to equalize the lengths of the probe path and improve the timeliness [137]. Packet timing analysis has also been investigated through a PTP-synchronized, clock-based INT implementation deployed on a multi-FPGA testbed [138]. Optimizations targeting service chains have also been proposed; for example, IntOpt minimizes probe overhead and delays for NFV service chains by optimally placing telemetry flows across programmable data planes [139]. Beyond overhead, INT has also been extended to advanced management tasks. Anomaly detection and congestion-aware routing were applied to INT metadata, as described in [140]. At the same time, a top-down model where operators issue high-level monitoring queries is automatically mapped to telemetry primitives was proposed in [141].

8.2.5 Network Flow Control

Flow control is a cornerstone of networking, shaping throughput, latency, and QoS. In SDN, the task is complicated by limited switch memory and heterogeneous link characteristics. To address this, the problem of joint rule caching and flow forwarding has been formulated as an end-to-end delay minimization challenge [142]. By decomposing the NP-hard problem into smaller subproblems, heuristic algorithms based on K-shortest paths, priority caching, and Lagrangian dual methods achieve roughly 20% delay reduction, underscoring the close interplay between rule management and flow control.

Beyond classical queuing models, recent work has explored flow control through several complementary directions. In cache-enabled networks, joint optimization of forwarding, caching, and control can reduce congestion and stabilize queues [143]. Within SDN, priority scheduling mechanisms dynamically adjust flow priorities to lower latency and ensure differentiated QoS, a capability expected to be vital in 6G environments [144]. Multimedia traffic introduces further complexity, where classification-based schemes supported by lightweight learning models improve responsiveness and reduce loss [145]. At the same time, hardware acceleration with FPGA-based SmartNICs demonstrates that real-time flow processing at line rate is achievable, extending flow control to high-performance programmable data planes [127].

8.3 Proposed Architecture, Automation, and Implementation

The packet processing pipeline under study is inspired by the P4THLS architecture [4], but redesigned and extended to implement the new proposed features. The underlying packet

processing architecture is elaborated. Modifications, optimizations, and new modules are described in detail. After explaining the backbone of the work, we walk through the new capabilities added to enrich metadata and support automatic conversion of P4 native stateful objects. Then, the integration of metadata and stateful objects into the FPGA-based data plane pipeline is presented, with a particular emphasis on seamless interaction across processing stages and intermodule interfaces.

The implementation of the new features proposed within the FPGA-based packet processing pipeline is organized into three stages:

1. The first stage introduces a methodology for per-packet metadata propagation, ensuring that contextual information can traverse the processing pipeline and be dynamically updated at each stage.
2. The second stage extends the pipeline with stateful objects that capture both flow-level and system-level dynamics, enabling persistent state management directly within the data path.
3. The third stage focuses on automating packet processing through an enhanced kernel that integrates baseline functionality with extended programmable capabilities.

Two representative use cases are implemented to demonstrate the effectiveness of these extensions:

1. In-band Network Telemetry (INT), where metadata propagation enables hop-by-hop visibility into network conditions;
2. Flow Control, where stateful objects provide congestion-aware behavior that adapts to real-time system dynamics.

8.3.1 Fundamental Packet Processing Architecture

Input and Stream Scheduling

Fig. 8.1 illustrates the high-level modules that comprise the packet processing pipeline and their interconnections within the FPGA. Four independent byte streams are merged into a single input stream by the input scheduler (IS), which applies a simple round-robin arbitration policy. The merging algorithm is not fixed and can be manually tuned or replaced by the user to introduce custom behaviors such as per-port prioritization or quality-of-service (QoS)

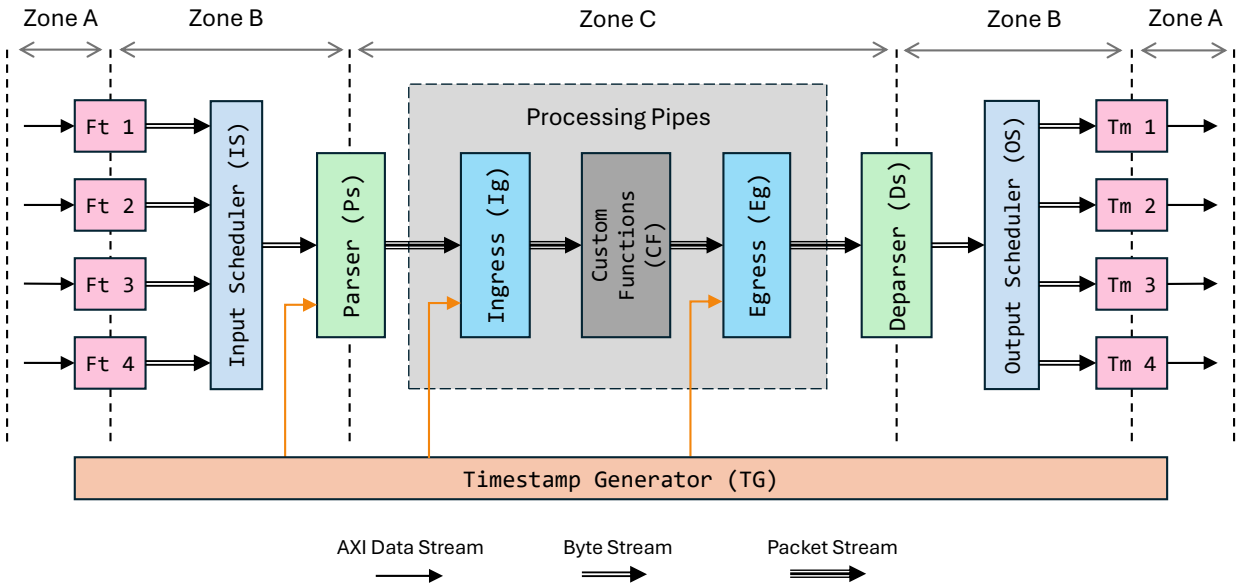

Figure 8.1 P4THLS packet processing pipeline [4] (extended with the proposed features) and inter-module interfaces. `Ft` and `Tm` denote the Fetch and Transmit blocks.

policies for packet delivery. Every receiver, or fetcher (Ft), implements its own logic to capture incoming packets and reformat them from AXI words into a contiguous byte stream.

The example pipeline shown supports four input ports and four output ports, though these values can be configured by the user at the time of P4-to-HLS conversion. A configuration with four inputs and outputs allows for direct connection of four QSFP28 lanes, each assigned to an independent receiver interface.

Parsing, Metadata, and Custom Extensions

Once merged, the byte stream is processed by the parser (Ps). The parser reconstructs the headers of the packets according to the flattened parse graph extracted from the JSON output of the P4 compiler. Upon successful parsing, the relevant metadata fields are assigned and attached to the packet. Among the metadata, timestamps form a critical group. To generate these timestamps, a free-running counter is implemented and shared between the parser, ingress, and egress stages. This counter ensures synchronized timing information without introducing dependencies that might stall the pipeline. Parsed packets are then buffered in a packet FIFO before entering the ingress block. Match-action processing is carried out within the ingress block. The processed packets are subsequently forwarded to the next pipeline stage.

Certain fixed stages, such as checksum verification defined in the `V1Model` [146], are excluded by default in this architecture to maintain efficiency in the base architecture. Nevertheless, the architecture allows users to incorporate such stages or even more complex functionality through custom extern modules. To this end, a custom function (CF) block is positioned between the ingress and egress stages, serving as a placeholder where user-defined modules can be inserted. Custom functions can be placed in any order in the processing pipeline (gray box region in Fig. 8.1). When such modules are added, the associated FIFOs and interfaces are automatically adapted to accommodate their placement in the processing pipeline.

Data Zones, Egress, and Deparsing

Three pipeline zones are defined to clarify the state of data and the associated FIFO structures at each stage. Zone A corresponds to AXI data stream interfaces, Zone B represents streams of bytes annotated with lane numbers and packet sizes, and Zone C denotes streams of packets enriched with metadata fields. These distinctions help characterize how information evolves as it traverses the processing pipeline and support precise timing and resource analysis across modules.

The egress block is retained for `V1Model` compatibility and to capture the egress timestamp, used to measure end-to-end latency from parser entry to egress. These measurements provide timing insight that allows the control plane to detect congestion and apply corrective actions. After egress processing, the deparser (Ds) reconstructs packet headers and payloads into a serialized stream, which the output scheduler (OS) dispatches to the proper egress ports via the transmit (Tm) modules.

Metadata for Packet Processing Pipelines

Metadata is attached to each packet at the parser stage, providing contextual information that accompanies it throughout the processing pipeline. It enables coordination between modules, consistent forwarding decisions, and precise performance monitoring. The structure largely follows the `V1Model` specification for compatibility with the P4C compiler, while additional fields (introduced during P4-to-HLS translation) extend functionality on the FPGA target. The complete metadata layout is shown in Listing 8.1.

- The `drop` field is a single-bit flag indicating whether a packet should be discarded. This flag is evaluated at the end of both the ingress and egress stages to prevent unnecessary processing of invalid or undesired packets.

Listing 8.1 Metadata fields

```
struct Standardmetadata {
  ap_uint<1> drop;
  ap_uint<3> ingress_port;
  ap_uint<3> egress_spec;
  ap_uint<3> egress_port;
  ap_uint<10> enq_qdepth;
  ap_uint<16> packet_length;
  ap_uint<64> parser_global_timestamp;
  ap_uint<64> ingress_global_timestamp;
  ap_uint<64> egress_global_timestamp;
  ap_uint<4> _padding;
};
```

- For scheduling and forwarding, three fields are retained for P4C compatibility: `ingress_port`, `egress_spec`, and `egress_port`. `ingress_port` identifies the input interface where the packet was received; `egress_spec` specifies the output port selected by the match–action logic; and `egress_port` indicates the final physical port resolved at the egress stage.
- The `enq_qdepth` field reports the occupancy of the input FIFO before the parser, reflecting the number of packets awaiting parsing. A low value indicates limited buffering and potential congestion. An output-side metric is omitted to prevent pipeline dependencies and throughput stalls.
- The `packet_length` field records the number of parsed bytes per packet and is updated immediately after successful parsing.
- Three global timestamps (`*_global_timestamp`, with `*` = `parser`, `ingress`, or `egress`) enable precise timing measurements. Captured at the entry of each stage, they provide fine-grained latency estimates across the pipeline. All are derived from a unified, free-running counter incremented every clock cycle for consistent timing system-wide.
- The `_padding` field is reserved for byte alignment and synthesis compatibility.

8.3.2 Stateful Object Support

The proposed solution can translate both stateless and stateful elements of a P4 program into the templated pipeline architecture. The corresponding control plane interface for stateful objects is also generated, enabling external configuration and management outside the FPGA. Generally, P4-native stateful objects can be grouped into three categories: (1) Registers; (2) Counters; and (3) Meters.

Listing 8.2 Sample P4 code fragment defining a direct counter

```
1  direct_counter(CounterType.packets) udp_d_counter;
2  table udp_port_firewall {
3      key = { hdr.udp.srcPort: exact;
4              hdr.udp.dstPort: exact;  }
5      actions = { forward;
6                  drop; }
7      size = 256;
8      default_action = drop();
9      counters = udp_d_counter;
10 }
```

Registers

Registers provide persistent storage in the data plane, retaining values across packet-processing operations. Those declared in the P4 program are automatically converted into HLS modules and inserted at the appropriate pipeline stage. As a guideline, each register should be read and written at most once per packet; additional accesses may introduce write-after-read or read-after-write hazards, causing functional or performance issues.

Although the framework does not strictly limit access frequency, excessive read/write operations can reduce throughput. Registers are also accessible from the control plane for external monitoring or updates. During conversion, users may select the underlying memory type: LUTRAM, Block RAM (BRAM), or Ultra RAM (URAM). Each register exposes two basic methods, `read()` and `write()`, for pipeline or control-plane interaction.

Counters

Counters track packet- or byte-level statistics, incrementing when specific conditions are met in the ingress or egress pipeline. They can operate as direct counters tied to table entries or as indirect counters managed independently. Values are readable asynchronously via the control plane.

Listing 8.2 shows a P4 snippet and its corresponding HLS implementation. The P4 fragment defines a direct counter `udp_d_counter` (line 1) associated with the table `udp_port_firewall` (lines 2–10). In P4, a direct counter is implicitly indexed by its table entry, so each entry has a dedicated counter automatically updated on every hit. The equivalent HLS code in Listing 8.3 illustrates how such a counter is translated into a parameterized C++ class that manages storage, updates, and access. Only the relevant parts of the generated code are shown for clarity.

Listing 8.3 The generated HLS code corresponding to the given P4 definition of a direct counter

```
1  enum CounterType {
2    PACKETS,
3    BYTES,
4    PACKETS_BYTES
5  };
6
7  template<unsigned int CSIZE, unsigned int CINDEX, unsigned int CWIDTH,
8           CounterType CTYPE> class Counter {
9  private:
10   ap_uint<CWIDTH> cnt_reg[CSIZE];
11 public:
12   Counter() { reset(); }
13   void count(const Packet &pkt, const ap_uint<CINDEX> index) {
14     if (index < CSIZE) {
15       ap_uint<CWIDTH> step = 0;
16       switch (CTYPE) {
17       case PACKETS: step = 1; break;
18       case BYTES: step = pkt.smeta.packet_length; break;
19       case PACKETS_BYTES : unsigned int shift = (CWIDTH + 6) >> 1;
20         step = ap_uint<CWIDTH>(1 << shift) |
21                ap_uint<CWIDTH>(pkt.smeta.packet_length);
22         break;
23       default: break; }
24       cnt_reg[index] += step;
25     }
26   }
27   ap_uint<CWIDTH> read(const ap_uint<CINDEX> index) {
28     return (index < CSIZE) ? cnt_reg[index] : 0;
29   }
30   void write(const ap_uint<CINDEX> index, const ap_uint<CWIDTH> value) {
31     if (index < CSIZE) { cnt_reg[index] = value; }
32   }
33 };
34
35 const unsigned int TS_Udp_Firewall_Table = 256;
36 const unsigned int CINDEX_udp_d_counter = 8;
37 const unsigned int CWIDTH_udp_d_counter = 64;
38 const CounterType TYPE_udp_d_counter = PACKETS;
39
40 class UdpPortFirewallTable<int TS_Udp_Firewall_Table, ...> {
41   Counter<TS_Udp_Firewall_Table, CINDEX_udp_d_counter,
42           CWIDTH_udp_d_counter, TYPE_udp_d_counter> icounter;
43   ...
44 };
```

The enumeration `CounterType` (lines 1-5) indicates whether the counter increments by packets, bytes, or a combination of both. The counter is implemented as a C++ template to support compile-time specialization (lines 7-8):

- `CSIZE`: the number of counter entries (corresponding to the table size in P4).
- `CINDEX`: the bitwidth of the index required to address the entries (computed as `ceil(log2(CSIZE))`).
- `CWIDTH`: the bitwidth of each counter cell (e.g., 64 bits).
- `CTYPE`: the counter type, taken from the enumeration `CounterType`.

Counters are stored in an array of templated-width unsigned integers `cnt_reg` (line 10), which can be mapped to different on-chip memory types depending on the given resource constraints. The constructor (line 12) initializes the counter array, the function `read()` (lines 27-29) returns the requested counter value, and the function `write()` (lines 30-32) allows the control plane to update or reset counters at runtime.

The function `count()` (lines 13-26) updates the counter for the specified index when a packet hits the associated table entry:

- `PACKETS`: increments by one per packet (line 17).
- `BYTES`: increments by the packet length (line 18).
- `PACKETS_BYTES`: combines both packet and byte counting. The counter is split so that the higher bits track the packet count (incremented by one per packet), while the lower bits track the total byte count (incremented by the packet length). Since the minimum Ethernet packet size is 64 bytes, we preserve 6 bits more ($2^6 = 64$) for the byte field, providing a balanced trade-off in bit allocation between packets and bytes (line 19-21). For instance, with `CWIDTH = 40`, 17 bits are assigned to packets (max $2^{17} - 1$), and 23 bits to bytes (max $2^{23} - 1$).

Template constants are automatically derived from the P4 program (lines 35–38). Each counter uses a default `CWIDTH` of 64 bits, customizable through a dedicated header file. The counter instance is embedded within the generated table class (lines 40–42), linking each table entry to its counter and updating it automatically on every table hit. Indirect counters follow the same mechanism, with their behavior translated from P4 semantics into the HLS environment during conversion.

Meters

Meters, on the other hand, implement token-bucket style mechanisms for traffic policing and shaping. In our architecture, we employ a simplified version of the RFC 2697 specification,

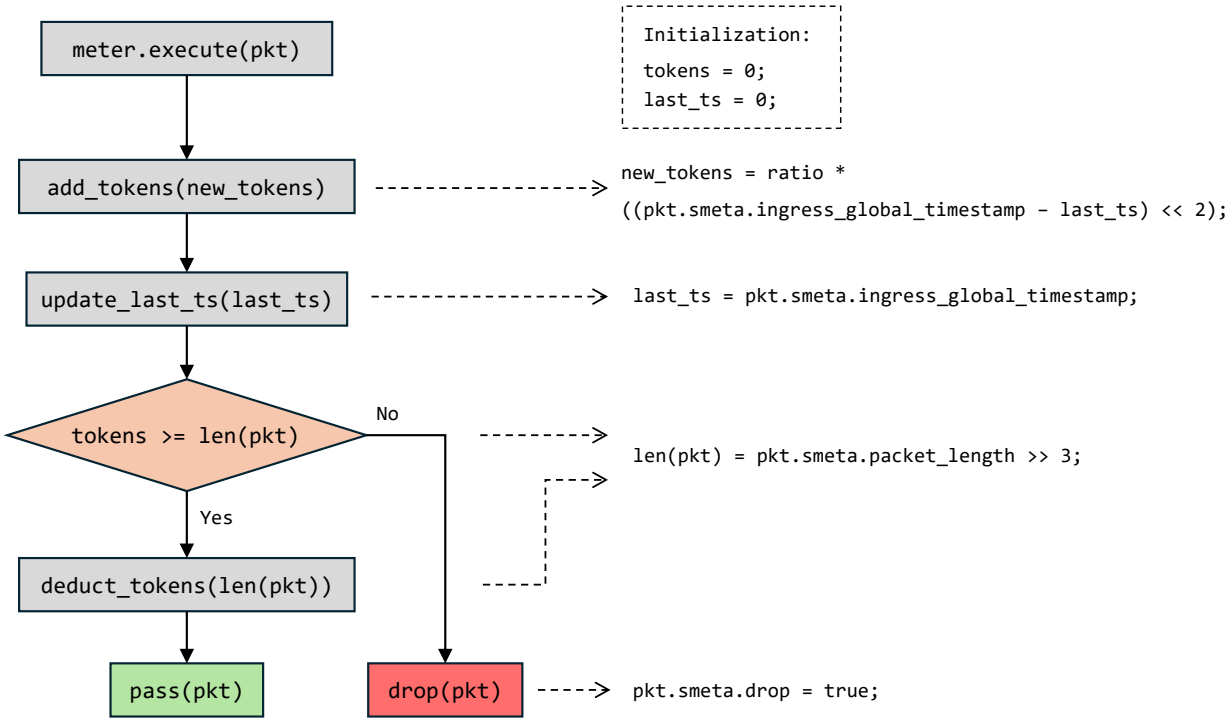

Figure 8.2 Meter algorithm using two colors and a single rate, assuming a 250 MHz clock frequency.

namely the single-rate two-color mechanism as performed in [5], for efficient hardware realization. Nevertheless, the implementation can be manually adapted to alternative policing methods if desired. Fig. 8.2 presents the flow chart of the metering process, which is described as follows:

- **Initialization**: The available tokens (`tokens`) and the last captured timestamp (`last_ts`) are initialized to zero at kernel startup.
- **Execution Trigger**: The function `meter.execute(pkt)` is invoked for each incoming packet to initiate the meter procedure.
- **Token Update**: The method `add_tokens(new_tokens)` replenishes the bucket based on elapsed time and the configured rate. New tokens are computed from the difference between the current packet timestamp and the stored one, left-shifted by two ($\times 4$) to match the 250 MHz clock, where each cycle equals 4 ns. This value is then scaled by a control-plane rate factor and saturated at a predefined limit to prevent overflow.
- **Timestamp Refresh**: The function `update_last_ts(ts)` updates the stored timestamp `last_ts` with the current packet's ingress timestamp. Depending on the desired

configuration, the timestamp may alternatively be derived from the parser stage.

- **Token Check and Decision**: The packet length is compared with the available tokens:
  - **Sufficient tokens**: The method `deduct_tokens(len(pkt))` decreases the token count by the packet size, and the packet is forwarded (`pass(pkt)`).
  - **Insufficient tokens**: The packet is marked for dropping by setting the field `pkt.smeta.drop` to `true`.

Like counters, meters can also be realized as direct or indirect objects. Their HLS translation follows the same methodology as that of counters presented in Listing 8.3. However, the implementation relies on dedicated functions defined in the meter flowchart.

8.3.3 Packet Processing Kernel Integration and Automation

The automated workflow for deploying P4 programs on FPGAs follows the same principles introduced in P4THLS, with additional extensions for stateful objects and metadata. The overall process is summarized in Fig. 8.3. The flow begins with a P4 source program, which is compiled using the BMv2 compiler (version 1.15.0) [105] targeting the `V1Model` architecture. This compilation produces a JSON description of the P4 pipeline. The JSON file, along with the original P4 code, is provided as input to the Code Generation Kernel, which serves as a Python-based transpiler. Following a templated pipeline architecture, this kernel translates the P4 constructs into equivalent HLS C++ modules. The translation process ensures that packet parsing, match-action execution, deparsing, and stateful object handling are all automatically mapped to synthesizable components.

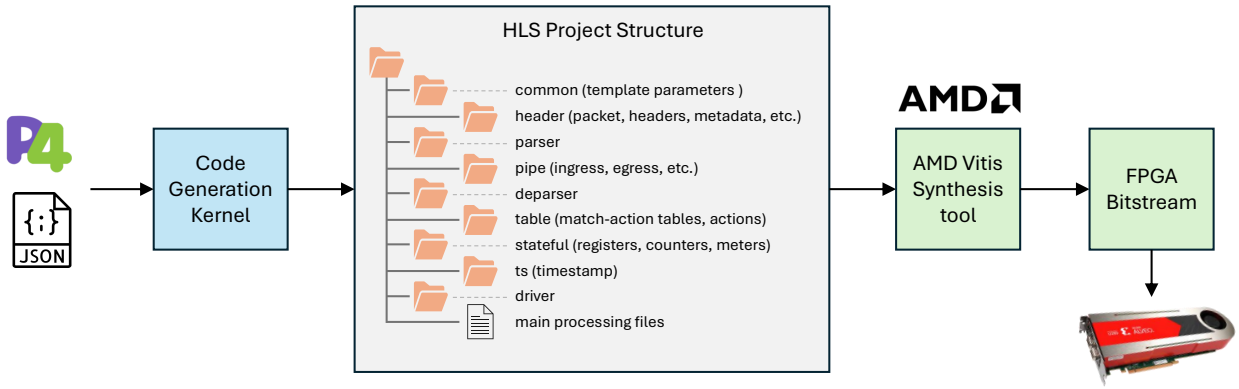

Figure 8.3 A simplified automation workflow: P4 to FPGA deployment (inspired by [4])

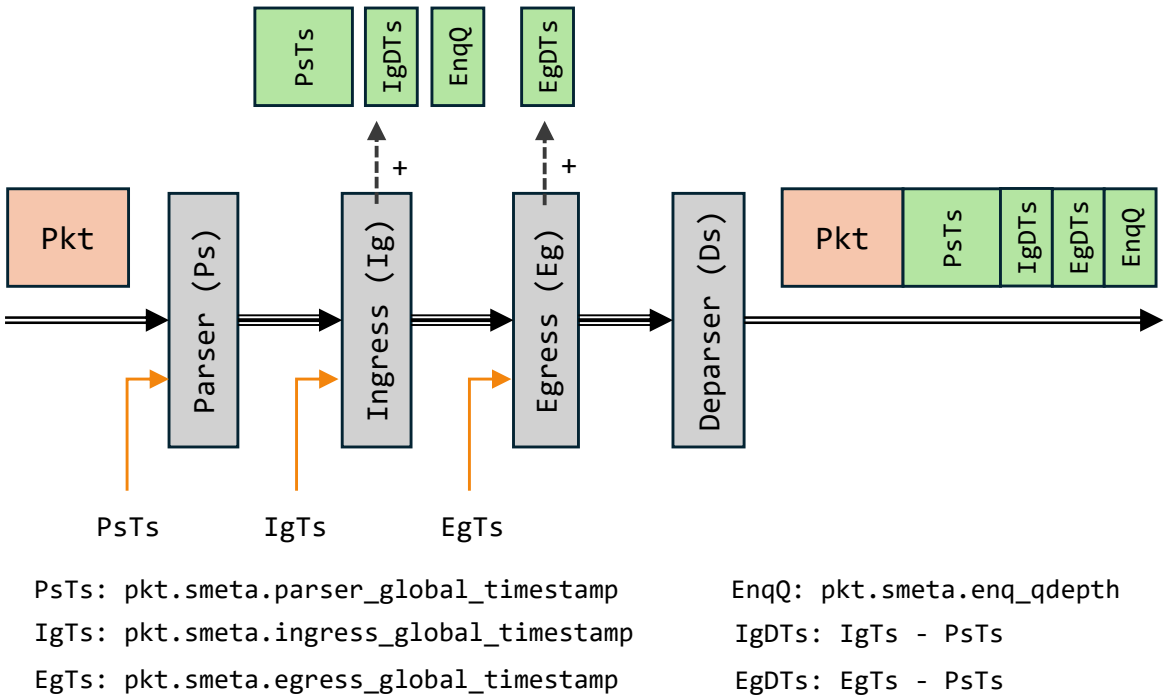

Figure 8.4 Custom header attachment for INT - PsTs: Parser Timestamp, IgDTs: Ingress Differential Timestamp, EgDTs: Egress Differential Timestamp, EnqQ: Input Queue Depth

The generated HLS project is then passed to the AMD Vitis HLS toolchain for high-level synthesis and FPGA implementation. This produces a deployable bitstream that instantiates the packet processing data plane directly on the FPGA. In parallel, the Code Generation Kernel generates host drivers and a control-plane interface to manage runtime configuration. These components enable the population of match-action tables and access to stateful objects through an AXI-Lite interface. Fig. 8.3 also shows the directory structure of the generated HLS project.

8.3.4 Architecture for In-band Network Telemetry

To demonstrate the extended functionality of the proposed pipeline, we implemented a network telemetry use case that leverages custom metadata headers to capture and export internal switch information. These headers enable the control plane to monitor pipeline timing and queueing behavior within the FPGA-based switch or NIC. Figure 8.4 illustrates the header stack appended to a packet, which includes the standard Ethernet, IP, and UDP headers, followed by custom INT headers and the original payload.

The custom INT header extends the packet with fields that expose timing and queueing infor-

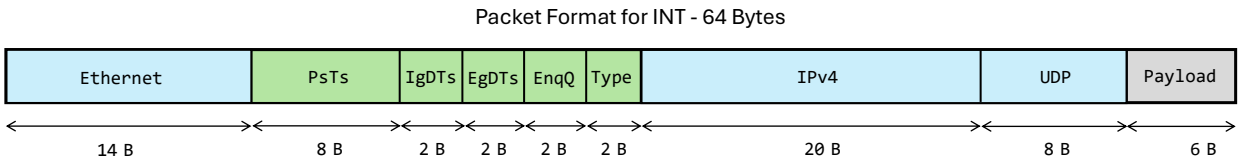

Figure 8.5 64-byte packet format for INT including custom headers - PsTs: Parser Timestamp, IgDTs: Ingress Differential Timestamp, EgDTs: Egress Differential Timestamp, EnqQ: Input Queue Depth

mation across the FPGA pipeline. These fields collectively provide a unified view of temporal and buffering behavior for each packet, enabling fine-grained performance monitoring. After the custom header is appended, the packet payload follows unchanged.

1. Parser Timestamp (`PsTs`): An 8-byte global reference captured at the parser stage and used as the baseline for subsequent measurements.
2. Ingress Delay Timestamp (`IgDTs`): A 2-byte value representing the delay between the parser and ingress stages.
3. Egress Delay Timestamp (`EgDTs`): A 2-byte value representing the delay between the ingress and egress stages. With 16-bit resolution, each field records up to 65,535 cycles, or about 262 μs at 250 MHz, sufficient for intra-switch delay measurement.
4. Enqueue Queue Depth (`EnqQ`): Reports the instantaneous occupancy of the input queue, reflecting short-term buffering conditions.

Fig. 8.5 shows the 64-byte packet format used to collect telemetry data from a single node. The packet begins with the Ethernet header, followed by a custom header carrying in-band telemetry information identified by a dedicated EtherType. The original EtherType is preserved within the custom header's Type field, allowing the remaining protocol stack and payload to be parsed normally. Each node appends or updates 16 bytes of local timing and queue-depth data before forwarding the packet. Time synchronization between nodes can be maintained via the Precision Time Protocol (PTP) or a centralized GPS reference [138].

8.3.5 Architecture for Flow Control

The integration of counters and meters into the FPGA dataplane extends functionality beyond simple packet classification and enables fine-grained flow control. Match-action tables,

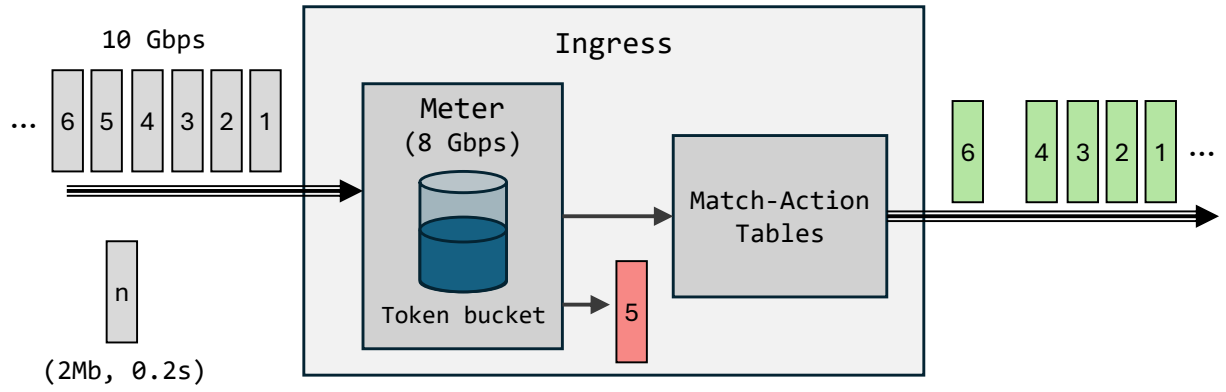

Figure 8.6 Meter mechanism to control the flow rate

by themselves, are limited to detecting and classifying flows based on predefined rules, such as marking flows as allowed or blocked. While this mechanism suffices for basic access control, it does not provide the capability to regulate traffic rates. Rate control typically requires frequent control-plane intervention, which is not performed in real time, introduces additional latency, consumes bandwidth between control and data planes, and ultimately results in slow and inaccurate control decisions.

In contrast, hardware-embedded meters enable automated and real-time enforcement of traffic rate policies directly in the dataplane. This feature is natively supported in the P4 language, but its architectural support varies. For example, software-based architectures such as v1model support meters but cannot deliver accurate rate enforcement due to limited timestamp precision and coarse-grained timing resolution. FPGA-based implementations, however, provide high-resolution timing using on-chip clock references, enabling packet and byte rate measurement with cycle-level accuracy. This precision is achieved by embedding timestamp mechanisms within the metadata carried by packets.

To demonstrate flow control, a single meter is instantiated at the ingress of the pipeline to regulate aggregate input traffic, as shown in Fig. 8.6. The meter is configured for a byte-rate limit of 8 Gbps and maintains a token bucket replenished according to timestamp differences. Upon packet arrival, the meter updates the token count using the ingress timestamp and compares it to the packet length. Packets are forwarded when sufficient tokens exist; otherwise, they are dropped. In this example, each 2 Mb traffic burst over 0.2 s forms a chunk. When the fifth chunk exceeds the 8 Gbps limit, its packets are dropped, while subsequent packets resume once enough tokens accumulate.

This mechanism can operate globally to control total ingress traffic or after match–action tables to enforce per-flow limits identified by a five-tuple or other header fields. By combining precise timestamping with hardware-accelerated meters, the proposed architecture enables real-time, accurate, and autonomous flow control within the dataplane, eliminating the need for frequent control-plane involvement.

8.4 Evaluation

8.4.1 Experimental Setup

This section evaluates the complete packet processing kernel using two representative network applications: 1) INT and 2) Flow control. The objective is to assess the flexibility and effectiveness of metadata handling and stateful object integration under realistic conditions, using performance metrics such as throughput, latency, accuracy, and FPGA resource utilization. In addition, the impact of the proposed features is evaluated against the baseline architecture to quantify their benefits and overhead.

All FPGA designs are synthesized using AMD Vitis HLS 2023.2 for high-level synthesis and analyzed with AMD Vivado 2023.2 after place-and-route. The AMD Alveo U280 FPGA card is the hardware platform, a data center–class accelerator optimized for high-throughput and low-latency networking workloads. Traffic generation and reception are performed via two Intel XXV710 Ethernet adapters, each providing 25 Gbps full-duplex operation. These adapters are directly connected to the FPGA's high-speed transceivers, enabling stress testing of the data plane at line rate. Synthetic traffic patterns are generated using the TRex traffic generator [74], which also captures and logs processed packets for performance analysis.

8.4.2 Comparison with Baseline

Table 8.1 compares the core P4THLS kernel against five extended variants of the architecture that incorporate metadata and stateful objects. The kernel variants are defined as follows:

- Baseline: A UDP firewall implemented over 256-bit AXI streams, consisting of two match-action tables with 256 entries each.
- Case 1: Baseline + metadata.
- Case 2: Case 1 + one array of 256 indirect counters (64-bit), mapped to BRAMs.
- Case 3: Case 1 + two arrays of 256 indirect counters (64-bit each), mapped to BRAMs.

- Case 4: Case 1 + one array of 256 indirect meters (single-rate, two-color), mapped to BRAMs.
- Case 5: Case 1 + two arrays of 256 indirect meters (single-rate, two-color), mapped to BRAMs.

Table 8.1 Comparison of the baseline P4THLS with different enriched stateful options (IC: Indirect Counter, IM: Indirect Meter)

Kernel Variant	Configuration	Resource Utilization				Max Frequency (MHz)	Max Throughput (Gbps)
		LUT (∼1.3M)	Flip-Flop (∼2.6M)	BRAM (2016)	URAM (960)		
Baseline	256-bit AXI bus + 2 MATs each with 256 entries	2034	4395	4	0	294.1	75.29
Case 1	Baseline + Metadata	2769	5139	4	0	288.4	73.83
Case 2	Case 1 + 1×256 of 64-bit IC	3564	6295	5	0	280.8	71.89
Case 3	Case 1 + 2×256 of 64-bit IC	3617	6313	6	0	281.2	71.99
Case 4	Case 1 + 1×256 IM	3549	6158	5	0	273.61	70.04
Case 5	Case 1 + 2×256 IM	3573	6176	6	0	271.92	69.61

Enabling metadata (Case 1) increases LUT utilization from 2,034 to 2,769 (≃36% overhead) and Flip-Flops from 4,395 to 5,139 (≃17%), without consuming additional memory blocks. Because metadata fields are embedded within the packet structure and traverse the pipeline, their hardware footprint remains minimal. The effect on operating frequency is negligible, causing only a slight throughput drop from 75.29 Gbps to 73.83 Gbps (≃1.5 Gbps difference). Introducing counters (Cases 2 and 3) increases both logic and memory use. A single 256-element counter array (64 bit each) adds 795 LUTs, 1,156 Flip-Flops, and one BRAM, reducing throughput by about 2 Gbps (73.83 to 71.99 Gbps). Adding a second array (Case 3)

incurs only minor additional cost—53 LUTs, 18 Flip-Flops, and one more BRAM. A similar pattern appears for meters (Cases 4 and 5). Each 256-entry meter array requires one BRAM, with logic overhead comparable to counters. Overall, the performance impact of adding counters or meters remains modest relative to Case 1.

Regarding memory mapping, the two match–action tables require four BRAMs in total, while each 256-entry counter or meter array occupies one BRAM. For instance, a 256-counter array with a 64-bit width needs about 16 Kb, which fits comfortably within a single 36 Kb BRAM. The additional logic required to integrate counters and meters into the pipeline remains modest, as many operations are shared across arrays. Address generation, access control, and control-plane interfaces are reused, with only minimal array-specific circuitry—such as offset registers and enable signals—added. Consequently, the main overhead of adding new arrays appears in BRAM usage, while the incremental logic cost is negligible. This trend is confirmed experimentally: adding a second 256-counter array required one additional BRAM but only a few extra LUTs and flip-flops. Overall, scaling to multiple stateful arrays follows a memory-dominated pattern, with logic overhead increasing very slowly.

8.4.3 Comparative Evaluation with Existing Work

Fig. 8.7 compares the metering accuracy of the proposed design with the method presented by Wang et al. [5] in target rates ranging from 100 to 700 Mbps. The proposed meter consistently achieves lower absolute relative error rates, demonstrating greater precision in rate regulation. At lower target rates such as 100 and 200 Mbps, the error is reduced from approximately 11.5% to 8.9% and from 6% to 3.75%, respectively. This improvement remains consistent with higher rates, where the error steadily decreases and remains below 3% for most cases. These results confirm that the proposed meter offers more stable and accurate rate enforcement across the entire range of target traffic rates.

The remaining inaccuracy in the proposed meter can be attributed to several implementation-level factors. First, both the time intervals and byte counts are represented as discrete integer values, introducing quantization effects in rate calculations. Additionally, the design employs a 10-bit left shift instead of a division by 1000 to simplify arithmetic operations and reduce hardware cost, which introduces minor coefficient approximations. Another contributing factor is the packet injection behavior of the traffic generator, where non-ideal scheduling and queueing delays can distort the inter-packet timing and queueing delays, slightly affecting the measured throughput accuracy.

Table 8.2 compares the utilization of FPGA resources for a stateful network use case designed to count TCP retransmission packets per flow. The operation compares the sequence

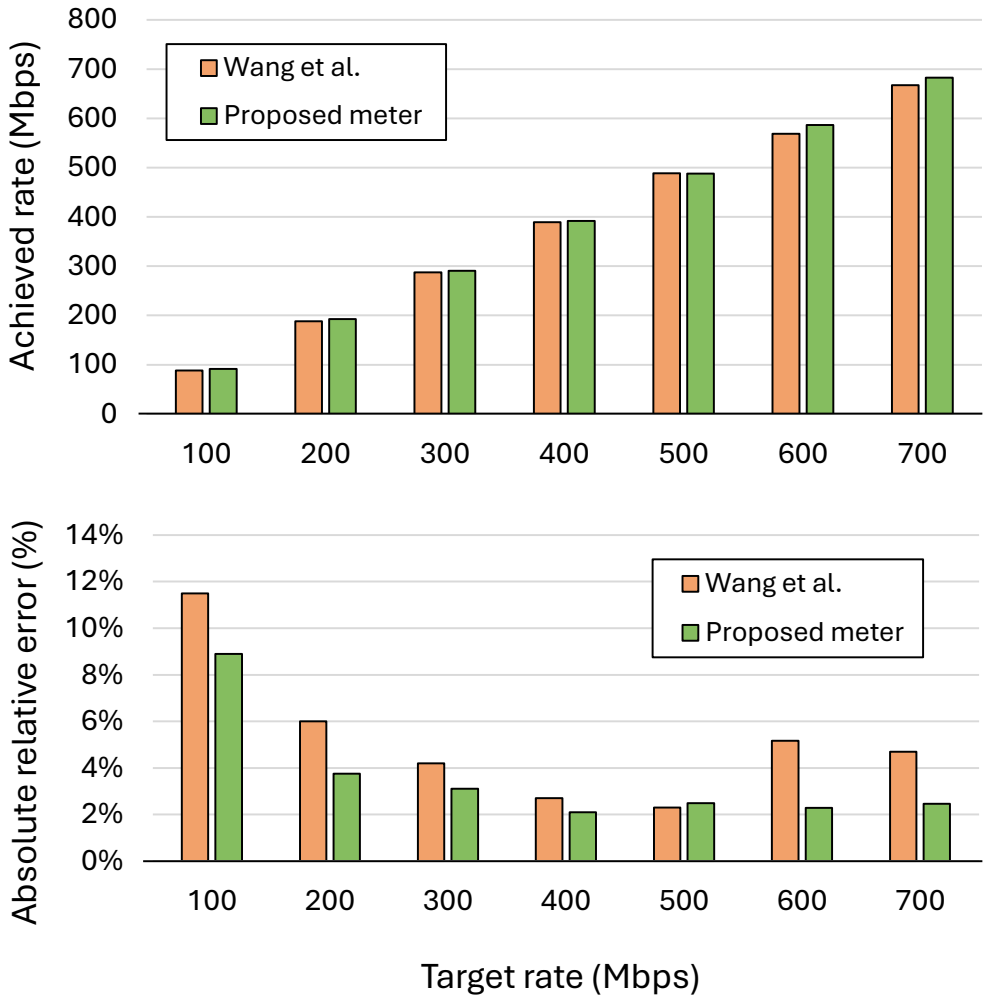

Figure 8.7 The achieved rate and absolute relative error to the target rate compared to Wang et al. [5].

number of each incoming packet with the stored per-flow sequence number. The scale of the implemented application, including the design parameters and table sizes, was carefully aligned with that of previous work to ensure a fair comparison. In the proposed design, a 32-entry TCAM with 256-bit width, taken from the HLSCAM [12] framework, is used along with the corresponding registers to maintain the sequence state for each flow.

Compared with existing FPGA-based stateful architectures, the proposed design substantially improves hardware efficiency. It uses only 14,374 LUTs, representing approximately a reduction of 5× reduction compared to FlowBlaze [123] and more than a 4× reduction relative to the method proposed by Lu et al. [121]. The proposed architecture also reduces the usage of Flip-Flop by about 2× compared to Lu's implementation. In terms of BRAMs, our

design consumes only 16 BRAMs; significantly less than FlowBlaze, whose baseline switch alone requires nearly twice this amount.

The reduced resource footprint can be attributed to two primary factors. First, the stateful logic in the proposed architecture is realized using compact, parameterized templates that closely integrate custom TCAM and register operations. This design minimizes redundant logic, control overhead, and wide interconnections within the data path. In contrast, existing approaches implement stateful behavior through a conventional match-action table through EFSMs, which, although conceptually consistent with the P4 data plane model, result in higher logic complexity. Second, the baseline switch in the proposed framework is derived from P4THLS, a lightweight and modular pipeline specifically optimized for FPGA synthesis. However, prior designs embed their stateful units into monolithic switch architectures that include extensive control, parsing, and matching logic, thus increasing the overall utilization and synthesis complexity.

8.5 Conclusion

This paper presented an extension to the pure HLS-based P4-to-FPGA compilation framework, P4THLS, enhancing its templated pipeline architecture to support metadata and stateful objects. The proposed extensions preserve the original advantages of P4THLS, direct translation from P4 descriptions into synthesizable HLS C++, while significantly improving the observability and functionality of the resulting FPGA-based data plane.

Metadata integration enables in-depth visibility into the internal operation of each FPGA node, supporting precise measurement of timing, queue depth, and other runtime parameters. This feature facilitates real-time monitoring and advanced telemetry applications such as INT. Moreover, the introduction of stateful objects, including registers, counters, and meters, expands the programmable data plane capabilities toward flow-aware and rate-aware processing. The implementation of a hardware-accurate meter demonstrates how these stateful extensions can be effectively combined with metadata to realize complex control mechanisms directly in the data plane.

Table 8.2 Comparison of FPGA resource utilization for a stateful network use case

Work	LUT	Flip-Flop	BRAM
FlowBlaze [123]	71,712	NA	393
Lu et al. [121]	60,871	35,898	NA
Proposed	14,374	18,293	16

Comprehensive evaluations validated the flexibility and efficiency of the proposed extensions. Compared to the baseline P4THLS kernel, the enhanced architecture maintains lightweight operation while offering expanded functionality. The reported experimental results confirmed that the proposed meter maintains the rate measurement errors below 3% at the 700 Mbps traffic rate. Furthermore, compared with existing FPGA-based stateful architectures, the proposed method achieves up to a 5× reduction in LUT utilization and a 2× reduction in Flip-Flop usage, highlighting its superior resource efficiency.

Overall, the proposed P4THLS extensions establish a unified and scalable foundation for developing advanced FPGA-based network functions that seamlessly combine visibility, control, and performance. The enhanced framework not only strengthens programmability and monitoring capabilities within FPGA-based data planes but also opens new opportunities for intelligent and adaptive network processing. Future work will focus on integrating the extended pipeline with AI-assisted mechanisms to enhance decision-making and in-network computing capabilities, as well as exploring distributed monitoring across multiple FPGA nodes to support large-scale, cooperative telemetry and control.

CHAPTER 9 GENERAL DISCUSSION

This dissertation has proposed a comprehensive and systematic pathway to integrate FPGAs into the programmable data-plane ecosystem, establishing a new paradigm where hardware programmability and language-level abstraction coexist seamlessly. The research was guided by the motivation to lower the traditional barriers of hardware design complexity while maintaining the deterministic performance and parallelism that make FPGAs unique. Through this endeavor, we presented a straightforward yet powerful methodology that brings FPGAs into the heart of data-plane architectures, forming a rich ecosystem for P4-based network programmability with minimal engineering overhead.

9.1 Interpretation of Results

The evaluation of the initial AGF implementation demonstrated the scalability of the proposed P4-to-FPGA data-plane design in supporting increasing numbers of user flows. The system was tested with 10k, 30k, and 50k flow entries, showing that overall FPGA resource utilization remained below 15%, with only URAM usage increasing proportionally to table size. This behavior indicates that the scalability of such applications is primarily constrained by the availability of on-chip memory resources rather than logic utilization.

P4THLS, as the heart of the research, demonstrated that this framework provides a highly efficient and customizable solution to map P4 applications onto FPGAs. Key performance indicators confirmed the framework's effectiveness in delivering high throughput and low latency, essential requirements for modern network infrastructures. The proposed framework was experimentally evaluated on an AMD Alveo U280 FPGA card to validate functionality, bandwidth, and end-to-end latency under realistic conditions. The setup used two Intel XXV710 25 Gbps Ethernet adapters connected directly to the FPGA's high-speed transceivers, with traffic generated and analyzed using the TRex Traffic Generator [74]. Within the FPGA, incoming packets were processed through the 10G/25G Ethernet Subsystem IP, converted to AXI-Stream format, and distributed across multiple lanes using a round-robin scheduler for balanced throughput. The host machine, interfaced via PCIe, managed control-plane operations including table configuration, counter monitoring, and runtime management. This setup ensured a data-center–grade evaluation environment capable of stress-testing the P4THLS pipeline at line-rate conditions. Performance evaluations showed that P4THLS achieved a maximum throughput of 143.3 Gbps with a 512-bit bus and 76.5 Gbps with a 256-bit bus. The framework demonstrated sub-50-cycle processing latency, with measured end-to-end la-

tencies consistently around 8 µs across various applications. These results indicate that the core templated HLS architecture, combined with configurable AXI bus widths, efficiently minimizes the cycles required for parsing and deparsing packets, thereby reducing overall latency.

Furthermore, the foundation of match-action tables in P4THLS is supported by the HLSCAM library, which offers fine-tuned HLS-based CAM implementations (BCAM and TCAM). The results of HLSCAM demonstrated that optimal modes, such as the High-Speed Enhanced (HS-H) mode, significantly improve efficiency, achieving up to a 7% reduction in LUT utilization and a 32.9% improvement in maximum operating frequency for TCAM configurations compared to the standard High-Speed mode. This high operating frequency could reach up to 31.18 Gbps for a 256×128 TCAM configuration.

The evaluation of the stateful extension in P4THLS revealed that the inclusion of metadata and stateful objects introduces only modest hardware and performance overheads. Enabling metadata increased LUT utilization by approximately 36% and Flip-Flops by 17%, yet required no additional memory resources, as metadata fields are embedded within the packet structure and flow naturally through the pipeline. The impact on performance was minimal, with throughput decreasing by only about 1.5 Gbps. Integrating counters and meters had a more pronounced effect on resource utilization due to their reliance on memory elements. A single array of 256 counters (64-bit each) required one BRAM, increasing logic usage by around 800 LUTs and 1,100 flipflops, while throughput declined by roughly 2 Gbps. Adding a second array produced a negligible additional logic cost, confirming that scaling follows a memory-dominated pattern. Each array of 256 counters or meters occupies one BRAM, leaving available capacity within the 36 Kb block, and the logic overhead remains small because control and access mechanisms are shared across arrays.

9.2 Comparison to Existing Literature

The initial work demonstrated that the proposed FPGA-based AGF achieves higher efficiency and scalability compared with existing P4-based BNG solutions. While prior implementations such as P4-BNG [21] and OpenBNG [22] support approximately 8k subscribers, and the design by Makhroute et al. [3] scales to over 400k flows at the cost of fully utilizing available SRAM and TCAM resources, the proposed FPGA design supports up to 50k user flows with less than 15% resource utilization and sustains an average latency of about 9 μs. These findings confirm that the FPGA-based approach offers a compact, low-latency, and resource-efficient solution that can scale effectively for practical AGF deployments.

The P4THLS framework demonstrates substantial advantages over existing FPGA-based P4 solutions, particularly concerning flexibility, performance, and resource efficiency. P4THLS generally outperforms commercial and academic counterparts in crucial metrics. In normalized throughput comparisons, P4THLS achieved rates of up to 51.4 Gbps operating at 256-bit AXI bus width, still higher than the performance of the commercial VitisNetP4 IP [57] (44.6 Gbps). P4THLS also exhibited significantly lower internal module latency (6–17 cycles) compared to VitisNetP4 (27–48 cycles) across various benchmarks, indicating a more efficient internal packet processing pipeline. Unlike proprietary tools like VitisNetP4, which use encrypted code, P4THLS is open-source and generates human-readable HLS C++ code, enabling easy customization and extension for advanced use cases.

The evaluation of the HLSCAM architecture confirms its high performance and scalability compared with existing FPGA-based CAM designs. The proposed implementations achieve higher or comparable operating frequencies relative to prior works such as Hierarchical TCAM [86], Zi-TCAM [87], HP-TCAM [38], and G-AETCAM [50], while supporting larger key widths and table depths. Although the operating frequency is slightly lower than D-TCAM [43] in some configurations, the 256×128 TCAM achieves up to 31.18 Gbps normalized throughput, indicating strong efficiency for wide search spaces. When compared with Xilinx CAM IPs, HLSCAM delivers consistently higher operating frequencies and throughput. The 256×128 and 512×36 TCAMs operate at 404 MHz and 365 MHz, outperforming the corresponding Xilinx IPs at 316 MHz and 323 MHz, respectively. This improvement yields normalized throughputs of 24.30 Gbps and 6.12 Gbps, compared with 19.04 Gbps and 5.47 Gbps. Although HLSCAM relies on distributed memory, leading to slightly higher CLB utilization than BRAM-based designs, it provides greater flexibility and overall superior performance across multiple configurations.

Incorporation of stateful objects into P4THLS yielded superior resource efficiency compared to prior FPGA-based stateful network architectures. For counting TCP retransmissions per flow needing stateful objects, the proposed architecture used only 14,374 LUTs, representing an approximate 5× reduction compared to FlowBlaze [123] and a 4× reduction compared to Lu et al [121]. Furthermore, Flipflop usage was reduced by about 2× compared to Lu's implementation. This efficiency stems from the use of compact, parameterized HLS templates for stateful logic, which minimizes redundant logic and contrasts with the higher complexity inherent in approaches based on Extended Finite State Machines (EFSMs) implemented through conventional match-action tables. The accuracy of the proposed single-rate two-color meter, leveraging the high-resolution timing of the FPGA platform, demonstrated greater precision than existing TCP-friendly meters reported by Wang et al. [5]. The proposed meter error remained below 3% at rates of around 700 Mbps, validating the precision enabled by

embedding timestamp mechanisms within packet metadata.

9.3 Addressing Research Objectives, Significance, and Implications

The central aim of this dissertation was to enable FPGA-based platforms to act as programmable data planes for P4 applications while maintaining flexibility, efficiency, and controllable performance. The research addressed this overarching goal through a progressive design methodology that systematically bridged the semantic abstraction of P4 with the architectural precision of FPGAs. The extent to which the research objectives were met can be assessed by reflecting on the theoretical foundations, architectural innovations, and experimental validations achieved throughout the work.

The first objective pursued the creation of a flexible methodology that repositions FPGAs as a readily deployable platform for network applications by identifying and addressing the architectural and methodological gaps that separate current programmable architectures from FPGA-oriented solutions. This objective was successfully realized through the conceptualization of a templated HLS-based methodology that eliminates the dependence on vendor-specific flows and proprietary IPs. By identifying limitations in previous P4-to-FPGA compilers, such as restricted language coverage, complex integration, and inadequate architectural visibility and flexibility, the research defined a clear design space in which high-level synthesis could serve as a bridge between programmability and performance. The methodology culminated in the development of the P4THLS framework, which provides a reproducible process to transform P4 programs into HLS-compatible hardware pipelines. This approach established a transparent workflow that allows hardware and network engineers to design, synthesize, and test FPGA-based packet processors without specialized RTL expertise.

The second objective focused on designing and implementing a flexible, high-performance, and configurable FPGA-based data plane capable of supporting P4 semantics. This goal was achieved through the development of a modular and templated pipeline architecture comprising parser, match-action, and deparser blocks, all automatically generated from P4 code. The HLSCAM library formed the foundation of this architecture, introducing parameterized binary and ternary CAMs implemented entirely in HLS. This innovation replaced vendor-specific IPs with a fully portable, open-source alternative, enabling precise control over memory depth, key width, and optimization strategy. Moreover, the architecture incorporated a unified memory interface supporting multiple on-chip resources and adaptable to external memory configurations, thereby ensuring scalability across different FPGA platforms. These elements fulfilled the second objective by proving that a fully P4-compatible, line-rate data-plane pipeline could be synthesized and deployed using a single, high-level

toolchain.

The third objective aimed to enable performance measurement, control, and evaluation across the FPGA-based data plane. This was realized through both architectural and experimental contributions. On the architectural side, the introduction of metadata propagation enriched the information carried with each packet, embedding contextual fields such as timestamps, packet size, queue occupancy, and port identifiers directly into the processing stream. This enhancement provided real-time visibility into pipeline behavior and facilitated performance monitoring without interrupting data-plane operations. In addition, the integration of stateful objects, including registers, counters, and meters, enabled the system to perform dynamic flow monitoring and feedback-based traffic regulation directly within the FPGA. The simplified single-rate two-color meter, based on RFC 2697, demonstrated practical feasibility for line-rate metering while maintaining deterministic timing characteristics.

The outcomes of this research integrate naturally with the broader discussion on the evolution of programmable data-plane architectures. While prior frameworks often traded flexibility for performance or relied on fixed RTL templates, this work introduces a synthesis methodology that retains the semantic richness of P4 while achieving efficient hardware realization. The discussion highlighted how abstraction, once considered a barrier to FPGA efficiency, can instead function as a powerful enabler when guided by carefully designed templates and compilation strategies. In this context, P4THLS stands not merely as a software tool but as a design philosophy: one that unifies the rapid prototyping capabilities of high-level programming with the precision and parallelism of FPGA hardware. It repositions the FPGA not as a specialized device requiring expert HDL design, but as a flexible, reconfigurable computing substrate accessible through a high-level, open-source workflow. In doing so, the research contributes to the broader movement toward software-defined hardware, advancing the field of network programmability and setting a precedent for similar cross-domain synthesis frameworks.

Collectively, the research fully achieved its objectives and, in several respects, surpassed them. The proposed architecture proved that FPGAs can function as efficient and flexible programmable data planes while introducing an automated, open workflow that simplifies design and deployment. The findings showed that abstraction and automation, when guided by structured templates, maintain and in some cases enhance performance and resource efficiency. Overall, the developed methodology establishes a robust foundation for translating P4 programs into high-performance FPGA implementations, advancing the state of programmable data-plane design for next-generation networks.

CHAPTER 10 CONCLUSION

10.1 Summary of Contributions

This dissertation presented a comprehensive research effort to bridge the gap between P4's high-level programmability and the hardware efficiency of FPGA-based architectures. The work proposed a straightforward and reproducible path to make FPGAs a natural and practical target for data-plane implementation, minimizing design complexity while maximizing flexibility and performance. Through the development of the P4THLS framework, this research introduced a unified and automated methodology that translates P4-defined packet-processing logic into HLS-based FPGA implementations. The resulting systems can be deployed directly through the Vitis design workflow, eliminating the need for manual RTL intervention.

The first contribution of this work is the development of templated Content-Addressable Memory implementations, realized entirely in HLS. To overcome the absence of CAM primitives in HLS-based design, this research proposed HLSCAM, a library that supports both binary (BCAM) and ternary (TCAM) architectures. The library enables parameterized exploration of table width, depth, and optimization strategies, allowing designers to balance resource utilization and latency while maintaining high operating frequencies. This contribution established a vendor-independent and easily reusable building block for high-speed packet classification in FPGA-based data planes.

The second contribution concerns the design of a flexible and unified data-plane architecture forming the core of the proposed framework. This templated FPGA pipeline integrates modular parser, match-action, and deparser stages connected through high-bandwidth AXI-Stream interfaces and a unified memory interface adaptable to different on-chip memory resources. This architecture provides scalability, modularity, and reusability across various P4-based applications while ensuring predictable performance.

The third contribution is the creation of an automated P4-to-HLS workflow, which serves as the foundation of the P4THLS framework. This workflow automatically translates P4 programs into human-readable HLS structures and generates all essential hardware modules, including parsers, match-action tables, and deparsers, along with compatible control-plane drivers. It enables researchers and engineers to design and deploy FPGA-based packet-processing pipelines from standard P4 source code with minimal manual intervention, establishing a complete toolchain that bridges software-defined networking and hardware acceler-

ation.

The fourth contribution is the construction of an integrated hardware and experimental testbed for systematic validation. The test environment is based on an AMD Alveo U280 FPGA accelerator, equipped with Intel network adapters and the TRex traffic generator. This setup allows evaluating throughput, latency, and scalability under realistic traffic conditions. The results demonstrate that the proposed framework sustains line-rate packet processing while maintaining low latency and efficient resource utilization.

The fifth and final contribution extends the framework with metadata and stateful object support, enabling more advanced packet-processing capabilities. By integrating registers, counters, and meters directly within the data path, the extended architecture supports complex functionalities such as telemetry, flow control, and traffic monitoring. Metadata fields, including packet size, timestamps, queue depth, and port identifiers, were introduced to enhance visibility into the internal pipeline, supporting in-band analysis and performance measurement. These additions allow advanced applications to operate entirely at line rate without relying on the control plane and represent an important step toward fully autonomous and intelligent data-plane systems.

The main outcomes of this research were distributed through a series of peer-reviewed publications, each addressing specific subsets of the dissertation's contributions:

1. Article 1: "Normal and Resilient Mode FPGA-Based Access Gateway Function Through P4-Generated RTL" (DRCN, 2024) [11]
    - Addresses the foundational feasibility study of mapping P4 programs to FPGAs through RTL generation.
    - Provides the initial proof of concept and identifies the design challenges and limitations that motivated the development of the P4THLS framework.
2. Article 2: "HLSCAM: Fine-Tuned HLS-Based Content Addressable Memory Implementation for Packet Processing on FPGA" (Electronics, 2025) [12]
    - Corresponds to the first contribution (Templated CAM Implementations).
    - Introduces the HLSCAM library, offering pure-HLS implementations of BCAM and TCAM modules with detailed design-space exploration and performance evaluation.
3. Article 3: "P4THLS: A Templated HLS Framework to Automate Efficient Mapping of P4 Data-Plane Applications to FPGAs" (IEEE Access, 2025) [4]

- Corresponds to the second, third, and fourth contributions (Unified Architecture, Automated Workflow, and Experimental Testbed).
- Presents the overall P4THLS architecture, automated toolchain, and system-level validation using the U280-based test environment.
- Demonstrates scalability and performance for multiple P4 applications.

4. Article 4: "A Stateful Extension to HLS-Based Data Plane Processing for Advanced Telemetry and Flow Control" (submitted to Future Internet, 2025)

   - Corresponds to Contribution 5 (Metadata and Stateful Extensions).
   - Extends the framework to integrate stateful objects and metadata propagation, enabling real-time telemetry and in-band flow management at line rate.
   - Demonstrates advanced application scenarios and performance enhancements.

10.2 Summary of Works

The research originated from an initial study that demonstrated the feasibility of translating P4 programs into FPGA hardware logic through RTL generation, using a real-world access gateway as a proof of concept. This early work provided critical insights into the design challenges and limitations of existing approaches and motivated the adoption of HLS as the primary methodology for developing a flexible, vendor-compatible, and easily extensible FPGA-based data-plane framework.

The research continued with the recognition that HLS offers an efficient abstraction for hardware design but can obscure low-level architectural visibility and provide limited support for domain-specific optimizations. To overcome these challenges, this work developed a templated architecture that encapsulates the essential elements of a programmable data plane. These elements include parsers, match-action tables, and deparsers, each synthesized from P4 code into modular HLS components. This structure simplified the design process while ensuring architectural consistency, timing determinism, and scalability across different use cases. These efforts led to the creation of the P4THLS framework, which provides a systematic and automated workflow for transforming P4 programs into synthesizable HLS code and enabling fully programmable and high-performance FPGA data planes. The main motivation for this research stemmed from the limitations of existing P4-to-FPGA approaches, which often suffer from proprietary dependencies, limited portability, and incomplete P4 language support. By addressing these challenges, this dissertation established a vendor-agnostic, open-source,

and extensible solution that allows researchers and engineers to deploy P4-defined packet-processing pipelines on reconfigurable hardware platforms in a straightforward and repeatable manner.

One of the most significant challenges encountered during this research was the implementation of match-action tables, which require fast and resource-intensive modules such as Content-Addressable Memories (CAMs). Such modules were not natively available in HLS. Although CAM modules exist as third-party intellectual property that can be instantiated in RTL-based designs, there had been no prior attempts to develop equivalent CAM structures using HLS. Since the goal of this work was to propose a fully HLS-based design that could be deployed on various FPGA devices and easily validated in simulation, without relying on external intellectual property, the HLSCAM library was introduced to address this gap. A major milestone of this dissertation, HLSCAM, provides parameterized implementations of both binary (BCAM) and ternary (TCAM) CAMs entirely within HLS. No prior work had achieved comparable results, and this contribution filled a crucial gap in the development of FPGA-based P4 pipelines. HLSCAM enables designers to explore trade-offs among table width, size, and speed while maintaining portability and avoiding the need for proprietary building blocks. The later introduction of a hash-based BCAM further enhanced lookup efficiency and broadened the design options within the P4THLS ecosystem.

After introducing P4THLS, the framework initially lacked support for several advanced features of the native P4 language and required additional architectural capabilities to handle more sophisticated applications. Building upon the P4THLS foundation, the framework was extended to support stateful objects and metadata propagation, which expanded its applicability to complex data-plane applications such as firewalls, flow control, and traffic monitoring. In the stateful extension, a practical scheme was developed to integrate registers, counters, and meters within the data path. Counters and meters can be defined as indirect, which allows manual management through the control plane, or as direct, which links them to specific tables for automatic updates during packet hits. This flexible mechanism enabled advanced functions to operate at line rate without involving the control plane, improving both responsiveness and throughput. The addition of metadata such as packet size, ingress and egress timestamps, queue occupancy, and input or output port identifiers introduced an introspective layer for real-time performance monitoring and analysis. Supporting these complementary features required structural modifications to the original packet-processing pipeline, particularly the separation of I/O-related blocks from processing blocks. This separation improved the overall stability of the pipeline by preventing stalls that could otherwise occur from I/O dependencies, thereby maintaining predictable and consistent performance under high-load conditions.

Through these cumulative developments, this dissertation delivers a complete, high-level, and open-source framework that simplifies the design of FPGA-based data planes. The results demonstrate that programmability and hardware efficiency can coexist effectively within modern network architectures when guided by structured design principles and systematic synthesis methodologies.

10.3 Limitations

While the use of HLS provides significant design flexibility and accelerates development, the resulting logic utilization in the P4THLS and HLSCAM implementations was occasionally less efficient than that achieved with carefully handcrafted RTL designs. This observation suggests that abstraction through HLS, although advantageous for productivity, introduces certain synthesis overheads that can affect resource efficiency. Further optimization within the HLS translation process would be beneficial to mitigate these inefficiencies and achieve finer-grained hardware optimization.

To preserve architectural generality and maintain synthesis efficiency, certain fixed stages defined in the v1model, such as checksum verification and validation blocks, were deliberately excluded from the default template pipeline. Although these functions can be integrated through user-defined extern modules, their inherent arithmetic complexity may impact overall processing rates and reduce achievable clock frequency. Incorporating these fixed verification stages without compromising latency or throughput will require additional architectural exploration and potentially deeper pipelining of these modules.

The proposed solution currently supports multiple on-chip memory resources through the unified memory interface but remains constrained when interfacing with off-chip memories such as DDR and HBM. Preliminary integration of external memory sources was functionally successful; however, performance degradation and occasional packet loss were observed at lower data rates due to the increased access latency of 100–300 ns inherent to these devices. Tolerating such latency requires larger FIFO buffers between pipeline stages, which, in turn, introduces significant resource overhead and affects timing closure. Moreover, DDR and HBM access latencies are non-deterministic, varying with controller scheduling and memory traffic, which complicates synchronization between packet data and corresponding table lookups. Addressing these challenges demands the introduction of a latency-tolerant mechanism capable of maintaining packet order and preventing loss.

One promising mitigation strategy would be the inclusion of a scratchpad memory or caching layer between the pipeline and external memory systems to reduce direct access latency. How-

ever, this approach introduces its own design complexity. Unlike CPU caches, which typically achieve hit rates above 90% due to strong spatial and temporal locality, network workloads often exhibit low locality, resulting in cache hit rates between 50% and 70%. Consequently, effective stall management and data prefetching mechanisms must be incorporated to sustain high throughput during cache misses and prevent pipeline stalls or packet drops. Designing such a mechanism remains an open challenge for future extensions of this architecture.

Finally, the meter module developed in this work implements a simplified single-rate two-color mechanism following the principles outlined in RFC 2697 [128], rather than adopting the full two-rate three-color marker (trTCM) model. This simplification was intentionally chosen to achieve efficient hardware synthesis and maintain predictable timing behavior across the processing pipeline. Nonetheless, the current design can be further extended to fully conform to the original RFC specification, incorporating both committed and peak information rate parameters to enable finer-grained traffic differentiation. The implemented meter also exhibits minor inaccuracies attributable to quantization effects, resulting from the discretization of time intervals and byte counts, as well as coefficient approximations introduced to reduce arithmetic complexity and resource utilization. These simplifications have a negligible impact under normal operating conditions but may lead to slight deviations in traffic classification accuracy during short burst scenarios.

10.4 Future Directions

While P4THLS establishes a robust and extensible foundation, several avenues remain open for further research and development. Future efforts should aim to extend language coverage by incorporating a broader range of built-in and user-defined extern objects, thereby enriching the architectural assets available to the framework. Supporting additional externs such as complex hashing units, advanced queue management mechanisms, and specialized counters would allow P4THLS to accommodate more sophisticated data-plane behaviors. This enhancement would bring the framework closer to full compliance with the evolving P4 standard and expand its usability across a wider spectrum of network applications.

From an architectural standpoint, the integration of off-chip memory subsystems, such as DDR and HBM, remains a challenging yet essential direction. The non-deterministic nature of external memory access introduces variability in latency that can disrupt the cycle-accurate timing of the packet-processing pipeline. Addressing this limitation requires innovative approaches such as hybrid caching, data prefetching, or scratchpad-based buffering to absorb latency fluctuations while maintaining deterministic performance. Exploring these mechanisms may also open new design paradigms for adaptive memory hierarchies in FPGA-based

network accelerators.

Another promising direction lies in expanding P4THLS toward multi-FPGA and heterogeneous deployments, enabling distributed or hierarchical data-plane architectures that can scale to terabit-level throughput. Such systems could partition complex pipelines across multiple reconfigurable devices or combine FPGAs with other accelerators, such as GPUs or DPUs, to create cooperative and workload-aware networking environments. This direction aligns closely with emerging trends in disaggregated and composable infrastructures, where reconfigurable hardware dynamically adapts to network conditions and performance demands.

Finally, a particularly impactful avenue involves the deployment of P4THLS in advanced network applications such as traffic reshaping, anomaly detection, and cybersecurity. Integrating the framework with machine learning algorithms capable of identifying complex traffic patterns and adaptive behaviors could significantly enhance the intelligence and responsiveness of FPGA-based network infrastructures. These application domains inherently require sophisticated processing pipelines that combine parsing, classification, metering, and policy enforcement at line rate. Leveraging the programmability and massive parallelism of FPGAs, P4THLS can evolve into a powerful enabler of in-network computing, where analytics, learning, and security functions are executed directly within the data path. Realizing this integration would mark a decisive step toward self-adaptive, intelligent FPGA-based data-plane architectures.

REFERENCES

[1] D. Scholz, H. Stubbe, S. Gallenmüller, and G. Carle, "Key properties of programmable data plane targets," in *2020 32nd International Teletraffic Congress (ITC 32)*, 2020, pp. 114–122.

[2] J. A. Brito, J. I. Moreno, L. M. Contreras, M. Alvarez-Campana, and M. Blanco Caamaño, "Programmable data plane applications in 5G and beyond architectures: A systematic review," *Sensors*, vol. 23, no. 15, 2023.

[3] E.-M. Makhroute, M.-A. Elharti, B. Vincent, Y. Savaria, and T. Ould-Bachir, "Implementing and evaluating a P4-based access gateway function on a Tofino switch," in *International Conference on Advanced Communication Technologies and Networking (CommNet)*, Dec. 2023, pp. 1–6.

[4] M. Abbasmollaei, T. Ould-Bachir, and Y. Savaria, "P4THLS: A templated hls framework to automate efficient mapping of p4 data-plane applications to fpgas," *IEEE Access*, vol. 13, pp. 164 829–164 845, 2025.

[5] S.-Y. Wang, H.-W. Hu, and Y.-B. Lin, "Design and implementation of TCP-friendly meters in P4 switches," *IEEE/ACM Transactions on Networking*, vol. 28, no. 4, pp. 1885–1898, 2020.

[6] E. F. Kfoury, J. Crichigno, and E. Bou-Harb, "An exhaustive survey on P4 programmable data plane switches: Taxonomy, applications, challenges, and future trends," *IEEE Access*, vol. 9, pp. 87 094–87 155, 2021.

[7] S. Kaur, K. Kumar, and N. Aggarwal, "A review on p4-programmable data planes: Architecture, research efforts, and future directions," *Computer Communications*, vol. 170, pp. 109–129, 2021.

[8] F. Hauser *et al.*, "A survey on data plane programming with P4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2023.

[9] O. Michel, R. Bifulco, G. Rétvári, and S. Schmid, "The programmable data plane: Abstractions, architectures, algorithms, and applications," *ACM Comput. Surv.*, vol. 54, no. 4, May 2021.

[10] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, 2014.

[11] M. Abbasmollaei, T. Ould-Bachir, and Y. Savaria, "Normal and resilient mode fpga-based access gateway function through p4-generated rtl," in *2024 20th International Conference on the Design of Reliable Communication Networks (DRCN)*, 2024, pp. 32–38.

[12] M. Abbasmollaei, T. Ould-Bachir, and Y. Savaria, "HLSCAM: Fine-Tuned HLS-Based Content Addressable Memory Implementation for Packet Processing on FPGA," *Electronics*, vol. 14, no. 9, 2025.

[13] P. Vörös *et al.*, "T4P4S: A target-independent compiler for protocol-independent packet processors," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018, pp. 1–8.

[14] P. Bosshart *et al.*, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *SIGCOMM Comput. Commun. Rev.*, vol. 43, no. 4, p. 99–110, aug 2013.

[15] C. Cascaval and D. Daly. (2017) P4 architectures. Accessed: Oct. 1, 2023. [Online]. Available: https://opennetworking.org/wp-content/uploads/2020/12/p4-ws-2017-p4-architectures.pdf

[16] I. Corporation. (2023) Intel tofino. Accessed: Dec. 1, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html

[17] M. W. Numan, B. J. Phillips, G. S. Puddy, and K. Falkner, "Towards automatic high-level code deployment on reconfigurable platforms: A survey of high-level synthesis tools and toolchains," *IEEE Access*, vol. 8, pp. 174 692–174 722, 2020.

[18] J. de Fine Licht, M. Besta, S. Meierhans, and T. Hoefler, "Transformations of high-level synthesis codes for high-performance computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 32, no. 5, pp. 1014–1029, 2021.

[19] V. A. Shirsath and M. M. Chandane, "Beyond the basics: An in-depth analysis and multidimensional survey of programmable switch in software-defined networking," *International Journal of Networked and Distributed Computing*, vol. 13, no. 1, p. 8, Dec 2024.

[20] F. Paolucci *et al.*, "P4 edge node enabling stateful traffic engineering and cyber security," *Journal of Optical Communications and Networking*, vol. 11, no. 1, pp. A84–A95, 2019.

[21] R. Kundel *et al.*, "P4-BNG: Central office network functions on programmable packet pipelines," in *2019 15th International Conference on Network and Service Management (CNSM)*, 2019, pp. 1–9.

[22] R. Kundel *et al.*, "OpenBNG: Central office network functions on programmable data plane hardware," *International Journal of Network Management*, vol. 31, no. 1, p. e2134, 2021.

[23] H. Harkous, M. Jarschel, M. He, R. Pries, and W. Kellerer, "P8: P4 with predictable packet processing performance," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2846–2859, 2021.

[24] A. Ebrahim, "High-level design optimizations for implementing data stream sketch frequency estimators on fpgas," *Electronics*, vol. 11, no. 15, 2022.

[25] A. Ebrahim, "Finding the top-k heavy hitters in data streams: A reconfigurable accelerator based on an fpga-optimized algorithm," *Electronics*, vol. 12, no. 11, 2023.

[26] M. Kekely and J. Korenek, "Mapping of p4 match action tables to fpga," in *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017, pp. 1–2.

[27] M. Kekely, L. Kekely, and J. Kořenek, "General memory efficient packet matching fpga architecture for future high-speed networks," *Microprocessors and Microsystems*, vol. 73, p. 102950, 2020.

[28] Y. Sun and Z. Guo, "The design of a dynamic configurable packet parser based on fpga," *Micromachines*, vol. 14, no. 8, 2023.

[29] C. Zeng *et al.*, "Tiara: A scalable and efficient hardware acceleration architecture for stateful layer-4 load balancing," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*. Renton, WA: USENIX Association, Apr. 2022, pp. 1345–1358.

[30] C. Beckmann, R. Krishnamoorthy, H. Wang, A. Lam, and C. Kim, "Hurdles for a dram-based match-action table," in *2020 23rd Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2020, pp. 13–16.

[31] N. Sultana *et al.*, "Flightplan: Dataplane disaggregation and placement for p4 programs," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, Apr. 2021, pp. 571–592.

[32] P. Bressana, N. Zilberman, D. Vucinic, and R. Soulé, "Trading latency for compute in the network," in *Proceedings of the Workshop on Network Application Integration/-CoDesign*, ser. NAI '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 35–40.

[33] R. Ricart-Sanchez, P. Malagon, A. Matencio-Escolar, J. M. Alcaraz Calero, and Q. Wang, "Toward hardware-accelerated QoS-aware 5G network slicing based on data plane programmability," *Trans. Emerg. Telecommun. Technol.*, vol. 31, no. 1, p. e3726, 2020.

[34] J. Rischke, C. Vielhaus, P. Sossalla, J. Wang, and F. H. Fitzek, "Comparison of upf acceleration technologies and their tail-latency for urllc," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2022, pp. 19–25.

[35] P4 DPDK Target. https://github.com/p4lang/p4-dpdk-target. Accessed: Jun. 1, 2024.

[36] D. Scholz *et al.*, "Cryptographic hashing in p4 data planes," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–6.

[37] L. Malina *et al.*, "Hardware-accelerated cryptography for software-defined networks with p4," in *Innovative Security Solutions for Information Technology and Communications*, D. Maimut, A.-G. Oprina, and D. Sauveron, Eds. Cham: Springer International Publishing, 2021, pp. 271–287.

[38] Z. Ullah, M. K. Jaiswal, and R. C. C. Cheung, "Design space explorations of hybrid-partitioned TCAM (HP-TCAM)," in *2013 23rd International Conference on Field programmable Logic and Applications*, 2013, pp. 1–4.

[39] Z. Ullah, M. K. Jaiswal, R. C. Cheung, and H. K. So, "UE-TCAM: An ultra efficient sram-based tcam," in *TENCON 2015 - 2015 IEEE Region 10 Conference*, 2015, pp. 1–6.

[40] A. M. S. Abdelhadi and G. G. F. Lemieux, "Deep and narrow binary content-addressable memories using fpga-based brams," in *2014 International Conference on Field-Programmable Technology (FPT)*, 2014, pp. 318–321.

[41] A. Qazi, Z. Ullah, and A. Hafeez, "Fast mapping and updating algorithms for a binary CAM on fpga," *IEEE Canadian Journal of Electrical and Computer Engineering*, vol. 44, no. 2, pp. 156–164, 2021.

[42] I. Ullah, Z. Ullah, U. Afzaal, and J.-A. Lee, "DURE: An energy- and resource-efficient TCAM architecture for fpgas with dynamic updates," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 27, no. 6, pp. 1298–1307, 2019.

[43] M. Irfan, Z. Ullah, and R. C. C. Cheung, "D-TCAM: A high-performance distributed ram based tcam architecture on fpgas," *IEEE Access*, vol. 7, pp. 96 060–96 069, 2019.

[44] Z. Shi *et al.*, "ME-TCAM: Memory-efficient ternary content addressable memory based on multipumping-enabled lutram on fpga," in *2020 International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI)*, 2020, pp. 38–42.

[45] M. Irfan, Z. Ullah, M. H. Chowdhury, and R. C. C. Cheung, "RPE-TCAM: Reconfigurable power-efficient ternary content-addressable memory on fpgas," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 8, pp. 1925–1929, 2020.

[46] A. Ahmed, K. Park, and S. Baeg, "Resource-efficient sram-based ternary content addressable memory," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 4, pp. 1583–1587, 2017.

[47] Q. Zou, N. Zhang, F. Guo, Q. Kong, and Z. Lv, "Multi-region sram-based TCAM fornbsp;longest prefix," in *Science of Cyber Security: 4th International Conference, SciSec 2022, Matsue, Japan, August 10–12, 2022, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2022, p. 437–452.

[48] N. Ur Rehman *et al.*, "Power efficient fpga-based tcam architecture by using segmented matchline strategy," in *2019 International Conference on Advances in the Emerging Computing Technologies (AECT)*, 2020, pp. 1–4.

[49] Z. Ullah, "LH-CAM: Logic-based higher performance binary cam architecture on fpga," *IEEE Embedded Systems Letters*, vol. 9, no. 2, pp. 29–32, 2017.

[50] M. Irfan and Z. Ullah, "G-AETCAM: Gate-based area-efficient ternary content-addressable memory on fpga," *IEEE Access*, vol. 5, pp. 20 785–20 790, 2017.

[51] D. M. and N. M. Sk, "TeRa: Ternary and range based packet classification engine," *Integration*, vol. 96, p. 102153, 2024.

[52] X. Song and Z. Guo, "An implementation of reconfigurable match table for fpga-based programmable switches," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 32, no. 11, pp. 2121–2134, 2024.

[53] S. Song, H. Choi, and H. Kim, "Fine-grained pipeline parallelization for network function programs," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 162–173.

[54] J. Huang *et al.*, "HAL: Hardware-assisted load balancing for energy-efficient snic-host cooperative computing," in *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, 2024, pp. 613–627.

[55] H. Eran, L. Zeno, Z. István, and M. Silberstein, "Design patterns for code reuse in hls packet processing pipelines," in *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2019, pp. 208–217.

[56] G. Karlos, H. Bal, and L. Wang, "Netcl: A unified programming framework for in-network computing," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '24. IEEE Press, 2024.

[57] Xilinx, "VitisNetP4: P4 language support for Xilinx devices," https://www.xilinx.com/products/intellectual-property/ef-di-vitisnetp4.html, 2024, accessed: 2024-10-05.

[58] I. Corporation, "Intel P4 suite overview for FPGA," https://www.intel.com/content/www/us/en/software/programmable/p4-suite-fpga/overview.html, 2024, accessed: 2024-10-05.

[59] P. Benácek, V. Pu, and H. Kubátová, "P4-to-VHDL: Automatic generation of 100 Gbps packet parsers," in *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 148–155.

[60] J. Santiago da Silva, F.-R. Boyer, and J. P. Langlois, "P4-compatible high-level synthesis of low latency 100 Gb/s streaming packet parsers in FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 147–152.

[61] H. Wang *et al.*, "P4FPGA: A rapid prototyping framework for P4," in *Proceedings of the Symposium on SDN Research*. New York, NY, USA: Association for Computing Machinery, 2017, p. 122–135.

[62] A. Yazdinejad, A. Bohlooli, and K. Jamshidi, "P4 to SDNet: Automatic generation of an efficient protocol-independent packet parser on reconfigurable hardware," in *International Conference on Computer and Knowledge Engineering (ICCKE)*, 2018, pp. 159–164.

[63] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The P4→NetFPGA workflow for line-rate packet processing," in *Proceedings of the ACM/SIGDA International Symposium on FPGA*. New York, NY, USA: ACM, 2019, p. 1–9.

[64] H. Wang *et al.*, "P4FPGA: A rapid prototyping framework for P4," in *Proceedings of the Symposium on SDN Research*, ser. SOSR '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 122–135.

[65] N. Zilberman, Y. Audzevich, G. A. Covington, and A. W. Moore, "NetFPGA SUME: Toward 100 Gbps as research commodity," *IEEE Micro*, vol. 34, no. 5, pp. 32–41, 2014.

[66] Z. Cao *et al.*, "P4 to FPGA-a fast approach for generating efficient network processors," *IEEE Access*, vol. 8, pp. 23 440–23 456, 2020.

[67] A. Yazdinejad, R. M. Parizi, A. Bohlooli, A. Dehghantanha, and K.-K. R. Choo, "A high-performance framework for a network programmable packet processor using P4 and FPGA," *Journal of Network and Computer Applications*, vol. 156, p. 102564, 2020.

[68] F. Leitão, R. David Carnero Ros, and J. Rius i Riu, "Fixed-mobile convergence towards the 5G era: Convergence 2.0: The past, present and future of FMC standardization," in *IEEE Conference on Standards for Communications and Networking (CSCN)*, 2016, pp. 1–6.

[69] BBF, "TR-456: AGF functional requirements, Issue: 02," BroadBand Forum, Tech. Rep., 2022.

[70] D. Lake, N. Wang, R. Tafazolli, and L. Samuel, "Softwarization of 5G networks–implications to open platforms and standardizations," *IEEE Access*, vol. 9, pp. 88 902–88 930, 2021.

[71] G. Brebner and W. Jiang, "High-speed packet processing using reconfigurable computing," *IEEE Micro*, vol. 34, no. 1, pp. 8–18, 2014.

[72] "AMD Adaptive Computing Documentation Portal — Vitis Networking P4 User Guide (UG1308)," https://docs.xilinx.com/r/en-US/ug1308-vitis-p4-user-guide/Target-Architecture, 2023.

[73] A. d. S. Ilha, C. Lapolli, J. A. Marques, and L. P. Gaspary, "Euclid: A fully in-network, p4-based approach for real-time ddos attack detection and mitigation," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 3121–3139, 2021.

[74] TRex: A Stateful Traffic Generator. GitHub Repository. Accessed: Sep. 1, 2023. [Online]. Available: https://github.com/cisco-system-traffic-generator/trex-core

[75] M. Irfan, K. Vipin, and R. Qureshi, "Accelerating DNA sequence analysis using content-addressable memory in fpgas," in *2023 IEEE 8th International Conference on Smart Cloud (SmartCloud)*, 2023, pp. 69–72.

[76] R. Kaplan, L. Yavits, R. Ginosar, and U. Weiser, "A resistive cam processing-in-storage architecture for DNA sequence alignment," *IEEE Micro*, vol. 37, no. 4, pp. 20–28, 2017.

[77] E. Garzón, L. Yavits, A. Teman, and M. Lanuzza, "Approximate content-addressable memories: A review," *Chips*, vol. 2, no. 2, pp. 70–82, 2023.

[78] M. Irfan, A. I. Sanka, Z. Ullah, and R. C. Cheung, "Reconfigurable content-addressable memory (CAM) on fpgas: A tutorial and survey," *Future Generation Computer Systems*, vol. 128, pp. 451–465, 2022.

[79] A. S. Krishnan, K. M. Sivalingam, G. Shami, M. Lyonnais, and R. Wilson, "Flow classification for network security using p4-based programmable data plane switches," in *2023 IEEE 9th International Conference on Network Softwarization (NetSoft)*, 2023, pp. 374–379.

[80] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: techniques, datasets and challenges," *Cybersecurity*, vol. 2, no. 1, p. 20, Jul 2019.

[81] Z. Xu, Z. Lu, and Z. Zhu, "Information-sensitive in-band network telemetry in p4-based programmable data plane," *IEEE/ACM Transactions on Networking*, vol. 32, no. 6, pp. 5081–5096, 2024.

[82] E. C. Paim and A. Schaeffer-Filho, "P4-turnet: Enabling nat traversal through p2p relay networks and programmable switches," in *2024 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2024, pp. 1–6.

[83] F. Ihle, S. Lindner, and M. Menth, "P4-PSFP: P4-based per-stream filtering and policing for time-sensitive networking," *IEEE Transactions on Network and Service Management*, vol. 21, no. 5, pp. 5273–5290, 2024.

[84] Z. Wen and G. Yan, "HiP4-UPF: Towards High-Performance comprehensive 5g user plane function on p4 programmable switches," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. Santa Clara, CA: USENIX Association, Jul. 2024, pp. 303–320.

[85] W. Jiang, "Scalable ternary content addressable memory implementation using fpgas," in *Architectures for Networking and Communications Systems*, 2013, pp. 71–82.

[86] Z. Qian and M. Margala, "Low power ram-based hierarchical CAM on fpga," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 2014, pp. 1–4.

[87] M. Irfan, Z. Ullah, and R. C. C. Cheung, "Zi-CAM: A power and resource efficient binary content-addressable memory on fpgas," *Electronics*, vol. 8, no. 5, 2019.

[88] Z. Ullah, M. K. Jaiswal, and R. C. C. Cheung, "Z-TCAM: An sram-based architecture for TCAM," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 23, no. 2, pp. 402–406, 2015.

[89] I. Ullah, Z. Ullah, and J.-A. Lee, "Efficient TCAM design based on multipumping-enabled multiported SRAM on fpga," *IEEE Access*, vol. 6, pp. 19 940–19 947, 2018.

[90] AMD/Xilinx Inc., *Binary CAM Search v2.6 LogiCORE IP Product Guide (PG317)*, v2.6 ed., AMD, Apr. 2023, vivado Design Suite 2023.2. [Online]. Available: https://docs.amd.com/r/2.6-English/pg317-bcam

[91] AMD/Xilinx Inc., *LogiCORE IP Ternary Content-Addressable Memory (TCAM) v2.6: Product Guide (PG318)*, v2.6 ed., AMD, Apr. 2023, vivado Design Suite 2023.2. [Online]. Available: https://docs.amd.com/r/2.6-English/pg318-tcam

[92] D. Kreutz *et al.*, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[93] J. A. Brito, J. I. Moreno, L. M. Contreras, M. Alvarez-Campana, and M. Blanco Caamaño, "Programmable data plane applications in 5G and beyond architectures: A systematic review," *Sensors*, vol. 23, no. 15, 2023.

[94] B. Yan *et al.*, "A survey of low-latency transmission strategies in software defined networking," *Computer Science Review*, vol. 40, p. 100386, 2021.

[95] S. Kianpisheh and T. Taleb, "A survey on in-network computing: Programmable data plane and technology specific applications," *IEEE Communications Surveys Tutorials*, vol. 25, no. 1, pp. 701–761, 2023.

[96] V. Chamola, S. Patra, N. Kumar, and M. Guizani, "FPGA for 5G: Re-configurable hardware for next generation communication," *IEEE Wireless Communications*, vol. 27, no. 3, pp. 140–147, 2020.

[97] G. C. Sankaran, K. M. Sivalingam, and H. Gondaliya, "P4 and NetFPGA-based secure in-network computing architecture for AI-enabled industrial internet of things," *IEEE Internet of Things Journal*, vol. 10, no. 4, pp. 2979–2994, 2023.

[98] S. Liu *et al.*, "In-network aggregation with transport transparency for distributed training," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 376–391.

[99] Y.-H. Lai *et al.*, "Heterocl: A multi-paradigm programming infrastructure for software-defined reconfigurable computing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 242–251.

[100] M. Ruiz, D. Sidler, G. Sutter, G. Alonso, and S. López-Buedo, "Limago: An FPGA-based open-source 100 GbE TCP/IP stack," in *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, 2019, pp. 286–292.

[101] D. Bansal *et al.*, "Disaggregating stateful network functions," in *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. Boston, MA: USENIX Association, Apr. 2023, pp. 1469–1487.

[102] S. K. Singh *et al.*, "Hybrid P4 programmable pipelines for 5G gNodeB and user plane functions," *IEEE Transactions on Mobile Computing*, vol. 22, no. 12, pp. 6921–6937, 2023.

[103] M. Ewais, J. C. Vega, A. Leon-Garcia, and P. Chow, "A framework integrating FPGAs in VNF networks," in *2021 12th International Conference on Network of the Future (NoF)*, 2021, pp. 1–9.

[104] M. Budiu and C. Dodd, "The P416 programming language," vol. 51, no. 1, p. 5–14, sep 2017.

[105] P. L. Consortium, "P4C: The P4 compiler," 2025, accessed: 2025-05-17. [Online]. Available: https://github.com/p4lang/p4c

[106] G. Fowler, L. C. Noll, K.-P. Vo, D. Eastlake, and T. Hansen, "The FNV non-cryptographic hash algorithm," in *IETF Meeting SECDISPATCH*, 2022, available online at https://datatracker.ietf.org/doc/html/draft-eastlake-fnv-18.

[107] R. Jenkins, "Hash functions: The good, the bad, and the ugly," *Dr. Dobb's Journal of Software Tools*, 1997, available online at https://www.drdobbs.com/jvm/algorithm-alley/184409859.

[108] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from Googles network infrastructure," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 58–72.

[109] Y. Li, Z. Ren, W. Li, X. Liu, and K. Chen, "Congestion control for AI workloads with message-level signaling," in *Proceedings of the 9th Asia-Pacific Workshop on Networking*, ser. APNET '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 59–65.

[110] B. Turkovic, S. Biswal, A. Vijay, A. Hüfner, and F. Kuipers, "P4QoS: QoS-based packet processing with P4," in *IEEE International Conference on Network Softwarization (NetSoft)*, 2021, pp. 216–220.

[111] A. G. Alcoz *et al.*, "Everything matters in programmable packet scheduling," in *Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI '25. USA: USENIX Association, 2025.

[112] T. Luinaud, T. Stimpfling, J. S. da Silva, Y. Savaria, and J. P. Langlois, "Bridging the gap: Fpgas as programmable switches," in *IEEE International Conference on High Performance Switching and Routing (HPSR)*, 2020, pp. 1–7.

[113] M. Nickel and D. Göhringer, "A survey on architectures, hardware acceleration and challenges for in-network computing," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 18, no. 1, Dec. 2024.

[114] E. D. Sozzo *et al.*, "Pushing the level of abstraction of digital system design: A survey on how to program fpgas," *ACM Comput. Surv.*, vol. 55, no. 5, Dec. 2022.

[115] B. Goswami, M. Kulkarni, and J. Paulose, "A survey on P4 challenges in software defined networks: P4 programming," *IEEE Access*, vol. 11, pp. 54 373–54 387, 2023.

[116] M. Jasny, L. Thostrup, T. Ziegler, and C. Binnig, "P4db - the case for in-network oltp," in *Proceedings of the 2022 International Conference on Management of Data*, ser. SIGMOD '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1375–1389.

[117] L. Castanheira, R. Parizotto, and A. E. Schaeffer-Filho, "FlowStalker: Comprehensive traffic flow monitoring on the data plane using P4," in *IEEE International Conference on Communications (ICC)*, 2019, pp. 1–6.

[118] X. Jin *et al.*, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*, ser. SOSP '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 121–136.

[119] D. Moro, D. Sanvito, and A. Capone, "FlowBlaze.p4: a library for quick prototyping of stateful SDN applications in P4," in *IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2020, pp. 95–99.

[120] A. Tulumello, "P4 language extensions for stateful packet processing," in *International Conference on Network and Service Management (CNSM)*, 2021, pp. 98–103.

[121] R. Lu and Z. Guo, "An FPGA-based high-performance stateful packet processing method," *Micromachines*, vol. 14, no. 11, 2023.

[122] F. Allard, T. Ould-Bachir, and Y. Savaria, "Enhancing P4 syntax to support extended finite state machines as native stateful objects," in *2024 IEEE 10th International Conference on Network Softwarization (NetSoft)*, 2024, pp. 326–330.

[123] S. Pontarelli *et al.*, "FlowBlaze: Stateful packet processing in hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. Boston, MA: USENIX Association, Feb. 2019, pp. 531–548.

[124] W. Feng *et al.*, "F3: Fast and flexible network telemetry with an FPGA coprocessor," *Proc. ACM Netw.*, vol. 2, no. CoNEXT4, Nov. 2024.

[125] Y. Chen, S. Layeghy, L. D. Manocchio, and M. Portmann, "P4-NIDS: High-performance network monitoring and intrusion detection in P4," in *Intelligent Computing*, K. Arai, Ed. Cham: Springer Nature Switzerland, 2025, pp. 355–373.

[126] B. M. Xavier, R. Silva Guimarães, G. Comarela, and M. Martinello, "Map4: A pragmatic framework for in-network machine learning traffic classification," *IEEE Transactions on Network and Service Management*, vol. 19, no. 4, pp. 4176–4188, 2022.

[127] Z. Han, A. Briasco-Stewart, M. Zink, and M. Leeser, "Extracting TCPIP headers at high speed for the anonymized network traffic graph challenge," in *2024 IEEE High Performance Extreme Computing Conference (HPEC)*, 2024, pp. 1–6.

[128] D. J. Heinanen and D. R. Guerin, "A Single Rate Three Color Marker," RFC 2697, Sep. 1999. [Online]. Available: https://www.rfc-editor.org/info/rfc2697

[129] D. J. Heinanen and D. R. Guerin, "A Two Rate Three Color Marker," RFC 2698, Sep. 1999. [Online]. Available: https://www.rfc-editor.org/info/rfc2698

[130] R. Laidig, F. Dürr, K. Rothermel, S. Wildhagen, and F. Allgöwer, "Dynamic deterministic quality of service model with behavior-adaptive latency bounds," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2023, pp. 127–136.

[131] L. Zhang, Y. Li, W. Yang, Q. Liu, and R. Yang, "PFTB: A prediction-based fair token bucket algorithm based on CRDT," in *International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2024, pp. 928–934.

[132] K. Liu *et al.*, "An anatomy of token-based congestion control," *IEEE Transactions on Networking*, vol. 33, no. 2, pp. 479–493, 2025.

[133] L. Kang, L. Tian, and Y. Hu, "A programmable packet scheduling method based on traffic rate," in *2024 IEEE 2nd International Conference on Control, Electronics and Computer Technology (ICCECT)*, 2024, pp. 211–215.

[134] L. Tan *et al.*, "In-band network telemetry: A survey," *Computer Networks*, vol. 186, p. 107763, 2021.

[135] S. Sardellitti *et al.*, "In Band Network Telemetry Overhead Reduction Based on Data Flows Sampling and Recovering," in *IEEE International Conference on Network Softwarization (NetSoft)*, 2023, pp. 414–419.

[136] Q. Zheng, S. Tang, B. Chen, and Z. Zhu, "Highly-efficient and adaptive network monitoring: When INT meets segment routing," *IEEE Transactions on Network and Service Management*, vol. 18, no. 3, pp. 2587–2597, 2021.

[137] Y. Zhang *et al.*, "INT-balance: In-band network-wide telemetry with balanced monitoring path planning," in *ICC 2023 - IEEE International Conference on Communications*, 2023, pp. 2351–2356.

[138] S. Bal *et al.*, "P4-based in-network telemetry for FPGAs in the open cloud testbed and FABRIC," in *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2024, pp. 1–6.

[139] D. Bhamare, A. Kassler, J. Vestin, M. A. Khoshkholghi, and J. Taheri, "IntOpt: In-band network telemetry optimization for NFV service chain monitoring," in *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, 2019, pp. 1–7.

[140] F. Li, Q. Yuan, T. Pan, X. Wang, and J. Cao, "MTU-Adaptive In-Band Network-Wide Telemetry," *IEEE/ACM Transactions on Networking*, vol. 32, no. 3, pp. 2315–2330, 2024.

[141] M. Yu, "Network telemetry: towards a top-down approach," *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 1, p. 11–17, Feb. 2019.

[142] L. Luo, R. Chai, Q. Yuan, J. Li, and C. Mei, "End-to-end delay minimization-based joint rule caching and flow forwarding algorithm for SDN," *IEEE Access*, vol. 8, pp. 145 227–145 241, 2020.

[143] B. Wei *et al.*, "Flow control oriented forwarding and caching in cache-enabled networks," *Journal of Network and Computer Applications*, vol. 196, p. 103248, 2021.

[144] Y.-F. Chen *et al.*, "Adaptive traffic control: OpenFlow-based prioritization strategies for achieving high quality of service in software-defined networking," *IEEE Transactions on Network and Service Management*, vol. 22, no. 3, pp. 2295–2310, 2025.

[145] P. S. Tiwana and J. Singh, "Enhancing multimedia forwarding in software- defined networks: An optimal flow mechanism approach," in *International Conference on Advancement in Computation Computer Technologies (InCACCT)*, 2024, pp. 888–893.

[146] The P4 Language Consortium. v1model.p4 – Architecture for simple_switch. https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4. Accessed: Oct. 1, 2025.