



Article

A Stateful Extension to P4THLS for Advanced Telemetry and Flow Control

Mostafa Abbasmollaei ^{1,*}, Tarek Ould-Bachir ^{1,*} and Yvon Savaria ^{2,*}

¹ MOTCE Laboratory, Department of Computer Engineering, Polytechnique Montréal, Montréal, QC H3T 1J4, Canada

² Department of Electrical Engineering, Polytechnique Montréal, Montréal, QC H3T 1J4, Canada

* Correspondence: mostafa.abbasmollaei@polymtl.ca (M.A.); tarek.ould-bachir@polymtl.ca (T.O.-B.); yvon.savaria@polymtl.ca (Y.S.)

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 incorporating INT support adds roughly 900 LUTs and 1000 Flip-Flops relative to the baseline switch. Furthermore, 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; stateful objects; In-band Network Telemetry; flow control



Academic Editors: Dania Marabissi and Lorenzo Mucchi

Received: 14 October 2025

Revised: 18 November 2025

Accepted: 19 November 2025

Published: 20 November 2025

Citation: Abbasmollaei, M.; Ould-Bachir, T.; Savaria, Y. A Stateful Extension to P4THLS for Advanced Telemetry and Flow Control. *Future Internet* **2025**, *17*, 530. <https://doi.org/10.3390/fi17110530>

Copyright: © 2025 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

Modern network infrastructures are increasingly dynamic and heterogeneous, supporting diverse applications that demand both high performance and adaptability [1]. Software-defined networking (SDN) has enabled programmability in the control plane; however, the data plane has remained largely constrained to simple match-action processing [2]. 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) [3], congestion-aware forwarding [4], and fine-grained quality of service (QoS) enforcement [5,6], 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 [7,8]. 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. 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 [9].

The P4 programming language has established itself as the de facto standard for describing programmable data plane behavior [10]. While P4-to-ASIC compilers 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 [11]. In [12], P4THLS was introduced as 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 in terms of stateful objects and structured metadata.

This paper presents a comprehensive architectural refactoring and major functional extension of the P4THLS framework. Essentially, we introduce native support for core P4 stateful entities, including registers, counters, and meters, along with a structured metadata propagation mechanism spanning all pipeline stages (All implementation codes are publicly available in the P4THLS repository at <https://github.com/abbasmollaei/P4THLS> (accessed on 15 November 2025)). This refactoring enhances the underlying architecture's modularity and extensibility, establishing a flexible, robust foundation that enables the seamless integration of future stateful and metadata-driven features.

The main contributions of this paper are summarized as follows:

1. Development of HLS-based implementations for P4 stateful objects, including registers, 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 a structured built-in metadata interface compatible with the V1Model architecture, encompassing standard fields such as timestamps, packet length, and I/O port identifiers, enabling consistent propagation of contextual information across all pipeline stages within the FPGA processing flow.
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 2 reviews background and related work on P4 programmability, stateful processing, and FPGA-based data plane design. Section 3 presents the proposed architecture, describing the extensions to P4THLS that incorporate structured metadata and provide native support for stateful objects. Section 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 5 concludes the paper and outlines future research directions.

2. Background and Related Work

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 [13], it has evolved into two main versions: P4₁₄ and P4₁₆. The language was designed with three main objectives in mind: reconfigurability, independence from specific protocols, and improved portability across different targets. Although complete portability cannot be guaranteed due to architecture-dependent features of each hardware, P4 allows high reuse of the same data-plane logic across various platforms [14].

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 [15].

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 have been proposed to enrich state management in the data plane, addressing this issue. Early work investigated scalable monitoring for traffic statistics [16] and in-network key-value caching [17]. FlowBlaze.p4 introduced a structured mapping of Extended Finite State Machines (EFSMs) [18], while subsequent efforts proposed broader stateful extensions [19,20]. More recently, EFSMs have been integrated as a native P4 construct, allowing the expression of complex behaviors [21].

Parallel interest has emerged in implementing stateful processing on programmable hardware. FlowBlaze was first deployed on an FPGA as a structured EFSM framework [22], inspiring hybrid systems that combine ASIC switching with FPGA co-processors for advanced telemetry [23]. 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 [24], while MAP4 demonstrates how ML classifiers can be mapped to programmable pipelines [25]. 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 [26].

Recent efforts have also targeted stateful processing in FPGA-based P4 data planes. Building on the P4THLS framework [12], 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 that FPGA-based SmartNICs programmed with P4 and HLS can sustain near-line-rate traffic analysis, showing that P4 abstractions can be combined with stateful extensions to support large-scale monitoring and telemetry [27]. Unlike these works, our contribution extends the P4THLS framework itself to natively support P4-style stateful objects and the propagation of structured metadata.

2.3. Meter Algorithms

Two widely adopted traffic metering algorithms are standardized in RFC 2697 [28] and RFC 2698 [29]. 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 [30]. The prediction-based fair token bucket coordinates distributed rate limiters, improving fairness and bandwidth utilization in cloud environments [31]. In addition, TCP-friendly meters have been implemented in P4 data planes, addressing the limitations of commercial switch meters and significantly improving TCP performance [32].

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 [33]. 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 [34].

2.4. In-Band Network Telemetry

In-band Network Telemetry (INT) embeds device and path state information directly into packets, enabling real-time, hop-by-hop network visibility without relying on external probes. INT supports three operational modes [35]: INT-XD (eXport Data), where nodes export telemetry metadata directly to a monitoring system without modifying packets; INT-MX (eMbed InstructXions), where the source node embeds INT instructions that guide each node to report metadata externally while keeping the packet size constant; and INT-MD (eMbed Data), the classic hop-by-hop mode in which both instructions and metadata are carried within the packet, and the sink node removes the accumulated telemetry information before forwarding. Consequently, network measurement has evolved from control-plane polling to data-plane telemetry [36], though the growing INT header size with path length can introduce overhead and affect 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% [37]. A proposed approach combines INT with segment routing to reuse header space and maintain a constant packet size [38]. At the same time, another approach introduced a balanced INT to equalize the lengths of the probe path and improve the timeliness [39]. Packet timing analysis has also been investigated through a PTP-synchronized, clock-based INT implementation deployed on a multi-FPGA testbed [40]. 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 [41]. 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 [42]. At the same time, a top-down model where operators issue high-level monitoring queries is automatically mapped to telemetry primitives was proposed in [43].

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 [44]. 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 [45]. 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 [46]. Multi-media traffic introduces further complexity, where classification-based schemes supported by lightweight learning models improve responsiveness and reduce loss [47]. 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 [27].

3. Proposed Architecture, Automation, and Implementation

The packet processing pipeline under study is inspired by the P4THLS architecture [12], 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.

3.1. Fundamental Packet Processing Architecture

3.1.1. Input and Stream Scheduling

Figure 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) 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.

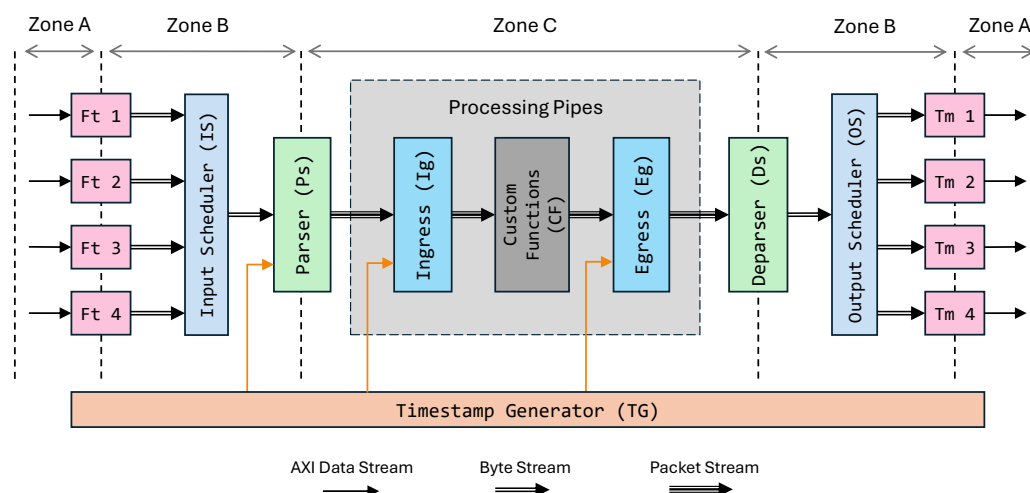


Figure 1. P4THLS packet processing pipeline [12] (extended with the proposed features) and inter-module interfaces. Ft and Tm denote the Fetch and Transmit blocks.

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.

3.1.2. Parsing, Metadata, and Custom Extensions

Once merged, the byte stream is processed by the parser (Ps). The parser reconstructs the packet headers 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.

The proposed architecture is inspired by both the V1Model [48] and XSA (AMD VitisNetP4) [49] architectures. The former preserves compatibility with a wide range of P4 use cases, while the latter aligns with realistic FPGA design constraints. Integrating certain fixed stages defined in the V1Model, such as checksum verification, would require additional pipeline tuning. Consequently, these stages are excluded from the base architecture to maintain hardware efficiency. Nonetheless, the design provides flexibility for users to incorporate such stages or even more complex functionality through custom extern modules. These custom functions are implemented manually in HLS and integrated into the main architecture through a dedicated Custom Function (CF) block positioned between the ingress and egress stages. The CF block serves as a placeholder where user-defined modules can be inserted in any order within the processing pipeline (gray box

region in Figure 1). When such modules are added, the associated FIFOs and interfaces are automatically adapted to accommodate their placement in the processing pipeline.

3.1.3. 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, supporting 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.

3.1.4. 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 metadata interface largely follows the V1Model specification for compatibility with the P4C compiler. This allows users to maintain consistent metadata field names in their P4 programs when deploying designs on the FPGA. The complete metadata layout is shown in Listing 1. The description of each field is provided below:

Listing 1. Metadata fields.

```

1 struct Standardmetadata {
2     ap_uint<1> drop;
3     ap_uint<3> ingress_port;
4     ap_uint<3> egress_spec;
5     ap_uint<3> egress_port;
6     ap_uint<10> enq_qdepth;
7     ap_uint<16> packet_length;
8     ap_uint<64> parser_global_timestamp;
9     ap_uint<64> ingress_global_timestamp;
10    ap_uint<64> egress_global_timestamp;
11    ap_uint<4> _padding;
12 };

```

- 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.
- 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 represents the final physical port, resolved only 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 a consistent timing system-wide.
- The `_padding` field is reserved for byte alignment and synthesis compatibility.

Except for the `parser_global_timestamp` and `drop` fields, all other metadata elements are inherited from the V1Model specification, providing high compatibility with standard P4 programs. The `mark_to_drop()` function can alternatively be used to set the drop bit. However, the `parser_global_timestamp` field becomes accessible only after the P4 code is converted into HLS.

Although previous studies such as FAST [50] and P4→NetFPGA [51] investigated built-in metadata in programmable pipelines, the proposed solution advances the concept by introducing multiple timestamp elements and defining metadata compatible with the V1Model architecture. This compatibility enhances portability, allowing P4 programs to be reused across different architectures with minimal changes. Additionally, the integration of stateful objects with built-in metadata enables practical applications such as flow control within an automated P4-to-HLS design flow.

Compared to the AMD VitisNetP4 metadata model [49], which defines a limited set of fields (`drop`, `ingress_timestamp`, `parsed_bytes`, and `parser_error`), the proposed metadata structure provides extended functionality (except for the `parser_error`). Additional fields such as `enq_qdepth`, `egress_spec`, and `egress_global_timestamp` enhance the observability of queue occupancy, forwarding behavior, and timing, respectively. These extensions allow tighter integration with the proposed stateful and telemetry mechanisms while maintaining compatibility with existing P4 use cases.

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.

3.2.1. 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.

3.2.2. 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 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 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 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 }
```

The enumeration `CounterType` (line 1) 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 2–3):

- `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 $\text{ceil}(\log_2(\text{CSIZE}))$).
- `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 5), which can be mapped to different on-chip memory types depending on the given resource constraints. The constructor (line 7) initializes the counter array, the function `read()` (lines 22–24) returns the requested counter value, and the function `write()` (lines 25–27) allows the control plane to update or reset counters at runtime.

The function `count()` (lines 8–21) updates the counter for the specified index when a packet hits the associated table entry:

- `PACKETS`: increments by one per packet (line 12).
- `BYTES`: increments by the packet length (line 13).
- `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 14–16). 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 29–32). 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 33–35), 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.

Listing 3. The generated HLS code corresponding to the given P4 definition of a direct counter.

```

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

```

3.2.3. Meters

Meters, on the other hand, implement token-bucket style mechanisms for traffic policing and shaping. In the proposed architecture, a simplified version of the RFC 2697 specification is employed, namely the single-rate two-color mechanism, as also described in [32], to enable efficient hardware realization. Using two colors, green (forward) and red (drop), together with a single threshold to monitor whether the traffic rate exceeds the limit, reduces the number of parameters and results in a simpler implementation suitable for typical flow control policies. Nevertheless, the implementation can be manually extended to support other policing methods, such as the standard srTCM (RFC 2697) or trTCM (RFC 2698). Figure 2 presents the flow chart of the metering process, which is described as follows:

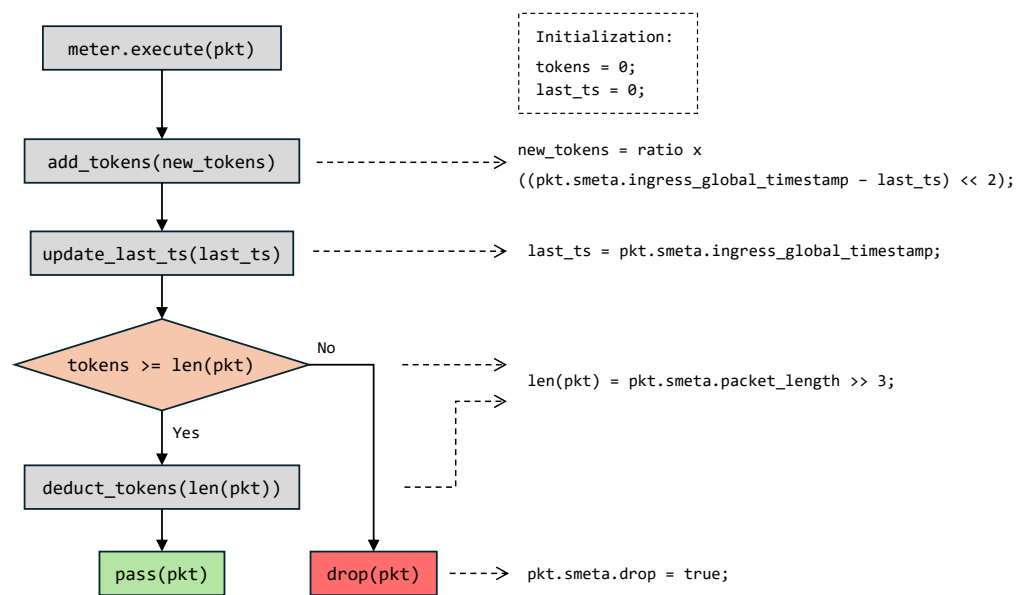


Figure 2. Meter algorithm using two colors and a single rate, assuming a 250 MHz clock frequency.

- **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 3. However, the implementation relies on dedicated functions defined in the meter flowchart.

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 Figure 3. The flow begins with a P4 source program, which is compiled using the BMv2 compiler (version 1.15.0) [52] 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 (Python version 3.11). 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.

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. Figure 3 also shows the directory structure of the generated HLS project.

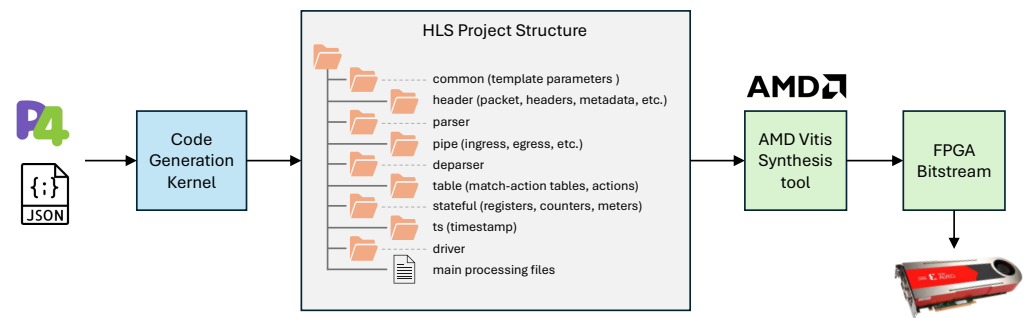


Figure 3. A simplified automation workflow: P4 to FPGA deployment (inspired by [12]).

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. Although the proposed architecture can operate under different INT modes depending on the user's implementation, this work primarily adopts the INT-MD mode. The context in this paper focuses on the source node implementation, while the transit and sink nodes can be defined similarly. Figure 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.

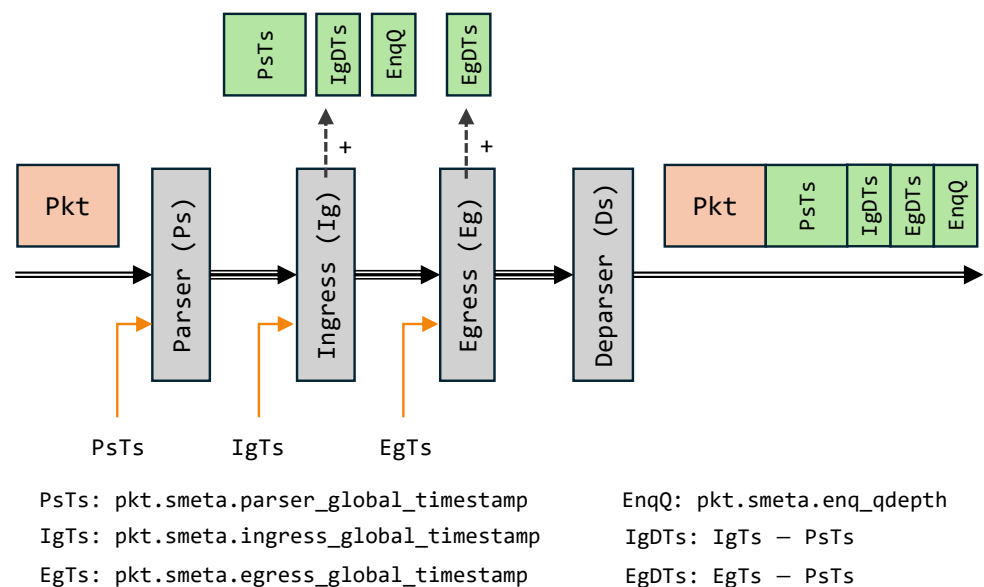


Figure 4. Custom header attachment for INT—PsTs: Parser Timestamp, IgDTs: Ingress Differential Timestamp, EgDTs: Egress Differential Timestamp, EnqQ: Input Queue Depth.

The custom INT header extends the packet with fields that expose timing and queueing information 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 μ s 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.

Figure 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 that carries 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 [40].

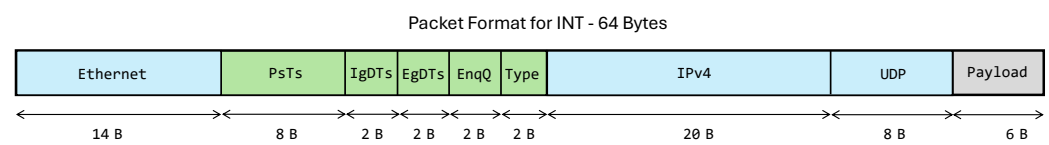


Figure 5. 64-byte packet format for INT including custom headers—PsTs: Parser Timestamp, IgDTs: Ingress Differential Timestamp, EgDTs: Egress Differential Timestamp, EnqQ: Input Queue Depth.

3.5. Architecture for Flow Control

The integration of counters and meters into the FPGA data plane extends functionality beyond simple packet classification, enabling fine-grained flow control. Match-action tables, on their own, are limited to detecting and classifying flows based on predefined rules, such as marking flows as either 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 data plane. 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 Figure 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 data plane, eliminating the need for frequent control-plane involvement.

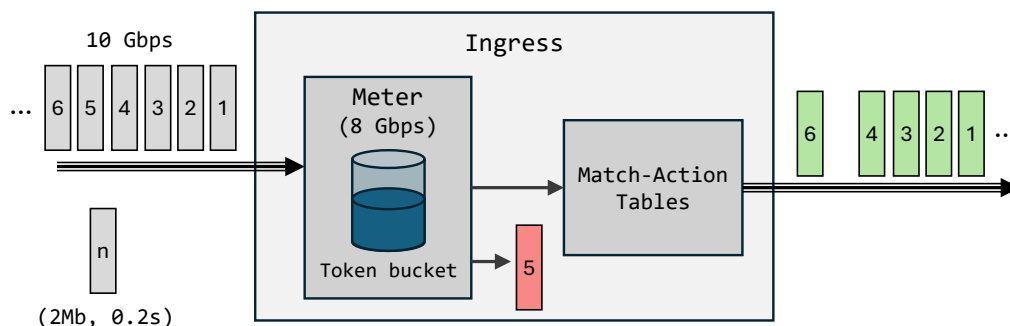


Figure 6. Meter mechanism to control the flow rate.

4. Evaluation

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. Additionally, the impact of the proposed features is evaluated against the baseline architecture to quantify their benefits and associated 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 capable of 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 [53], which also captures and logs processed packets for performance analysis.

4.2. Comparison with Baseline

Table 1 compares the baseline P4THLS switch with the proposed INT-enabled variant. Enabling INT increases LUT usage from 2034 to 2912 (~900) and Flip-Flops from 4395 to 5371 (~1000) due to the additional metadata management logic across the pipeline and the inclusion of telemetry information within packets. BRAM and URAM utilization remain unchanged, as the extra buffering required is implemented using Flip-Flop resources. The maximum clock frequency decreases slightly, resulting in a proportional drop in peak throughput from 75.29 to 72.42 Gbps. The average end-to-end latency remains at 8.1 μ s, with only a small increase in standard deviation from 1.34 to 1.41. The similar latency values observed in both implementations occur because the total end-to-end delay is primarily governed by external factors such as link transmission, signal propagation, and queuing within the I/O subsystems. These factors outweigh the minor differences in internal pipeline processing time, making them negligible in the overall latency measurement.

Table 1. Comparison between the baseline P4THLS switch and the proposed INT-enabled switch.

Kernel Variant	Resource Utilization				Max Frequency (MHz)	Max Throughput (Gbps)	End-to-End Latency (μ s) Std. Dev. (σ)
	LUT ($\sim$ 1.3 M)	Flip-Flop ($\sim$ 2.6 M)	BRAM (2016)	URAM (960)			
Baseline Switch (P4THLS) [12]	2034	4395	4	0	294.1	75.29	8.1/ $\pm$ 1.34
INT-enabled Switch (proposed)	2912	5371	4	0	282.91	72.42	8.1/ $\pm$ 1.41

Table 2 compares the core P4THLS kernel against five extended variants of the architecture that incorporate metadata and stateful objects. The baseline in this evaluation corresponds to the original P4THLS framework [12], whereas the variants employ the enhanced architecture proposed in this work. 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.

Enabling metadata (Case 1) increases LUT utilization from 2034 to 2769 ($\simeq$ 36% overhead) and Flip-Flops from 4395 to 5139 ($\simeq$ 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 ($\simeq$ 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, 1156 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, requiring 53 LUTs, 18 Flip-Flops, and one additional 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.

The increase in resource utilization primarily arises from the additional registers and control logic required to manage the propagation of metadata throughout the entire pipeline and to handle stateful objects, such as counters and meters. These supplementary modules require additional buffering capacity, dedicated arithmetic units (performing operations such as addition, shifts, and comparisons), and extended control pathways to maintain the per-flow state persistently.

Regarding memory mapping, the two match–action tables require a total of four BRAMs, while each 256-entry counter or meter array occupies one BRAM. For instance, a 256-counter array with a 64-bit width requires approximately 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 being 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.

Table 2. 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 (~1.3 M)	Flip-Flop (~2.6 M)	BRAM (2016)	URAM (960)		
Baseline Switch (P4THLS) [12]	256-bit AXI bus + 2 MATs each with 256 entries	2034	4395	4	0	294.1	75.29
Case 1	Baseline (proposed) + 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

4.3. Comparative Evaluation with Existing Work

Figure 7 compares the metering accuracy of the proposed design with the method presented by Wang et al. [32] in target rates ranging from 100 to 700 Mbps. The comparison rate was limited to 700 Mb/s to align with the results reported in [32], as no higher-rate data were available from that study. Nevertheless, the proposed pipeline can sustain higher traffic rates, and no clear saturation or limiting throughput was observed in the experiments, indicating that the meters and processing stages remain fully functional beyond the tested rate. 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, which can introduce non-ideal scheduling and queueing delays that distort the inter-packet timing and affect the measured throughput accuracy.

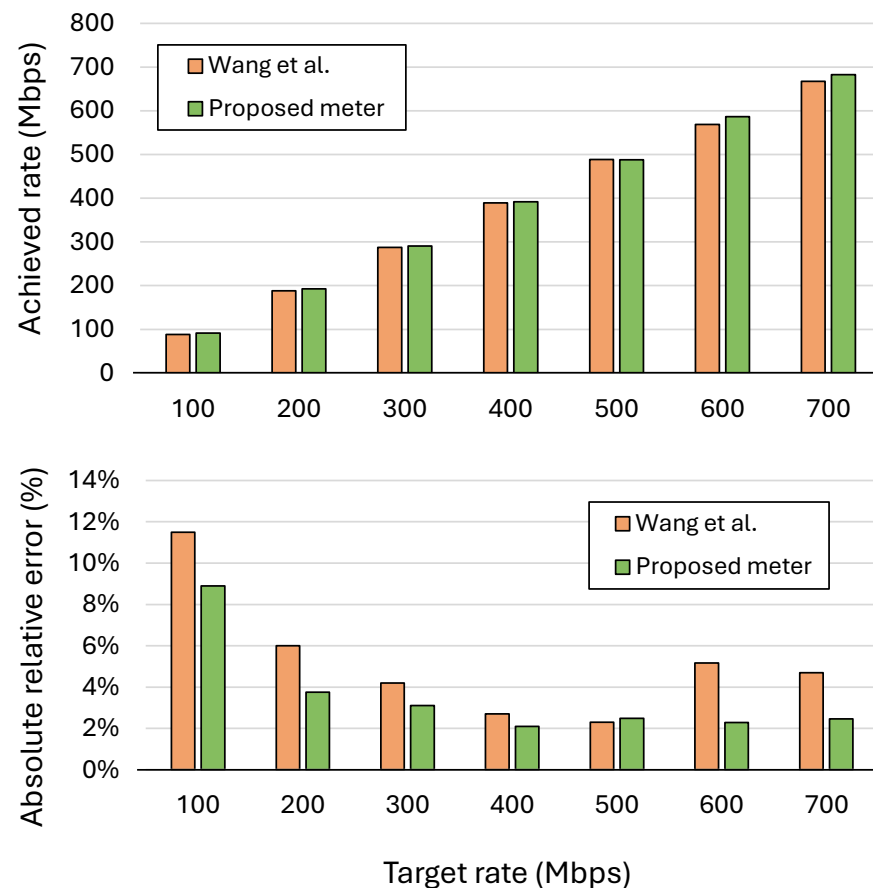


Figure 7. The achieved rate and absolute relative error to the target rate compared to Wang et al. [32].

Table 3 compares the experimental parameters and the utilization of FPGA resources for a stateful network use case designed to count TCP retransmission packets per flow. The operation compares the sequence 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 [54] 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. The implementation uses an Ultrascale+ FPGA running at 250 MHz, whereas FlowBlaze [22] was realized on a NetFPGA SUME board operating at 156.25 MHz, and Lu et al. [20] employed a Zynq Ultrascale+ device at 200 MHz. Despite operating at a higher frequency, the proposed design uses only 14,374 LUTs, achieving about a 5× reduction compared to FlowBlaze and more than a 4× reduction relative to Lu’s design. It also reduces Flip-Flop usage by approximately 2× compared with Lu’s implementation. In terms of BRAM utilization, the proposed design consumes only 16 blocks, which is significantly fewer than the 393 BRAMs of FlowBlaze.

Table 3. Comparison of FPGA resource utilization for a stateful network use case.

Work	FPGA	Clock Freq.	LUT	Flip-Flop	BRAM
FlowBlaze [22]	Virtex 7 (NetFPGA SUME)	156.25 MHz	71,712	NA	393
Lu et al. [20]	Zynq Ultrascale+	200 MHz	60,871	35,898	NA
Proposed	Ultrascale+ (Alveo U280)	250 MHz	14,374	18,293	16

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 conventional match-action tables using 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.

5. Conclusions

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.

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. On the AMD Alveo U280 FPGA, the integration of INT functionality increases resource usage by only about 900 LUTs and 1000 Flip-Flops compared to the baseline switch, demonstrating that telemetry support can be achieved with minimal hardware overhead. 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 $5\times$ reduction in LUT utilization and a $2\times$ 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. Additional fixed functional blocks, such as checksum verification and CRC hashing, are also planned for integration into future versions of the architecture.

Author Contributions: Conceptualization, M.A. and T.O.-B.; methodology, M.A., T.O.-B. and Y.S.; software, M.A.; validation, M.A., T.O.-B. and Y.S.; formal analysis, M.A., T.O.-B. and Y.S.; investigation, M.A., T.O.-B. and Y.S.; resources, M.A.; data curation, M.A.; writing—original draft preparation, M.A. and T.O.-B.; writing—review and editing, M.A., T.O.-B. and Y.S.; visualization, M.A.; supervision, T.O.-B. and Y.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by NSERC grant number IRCPJ-548237-18 CRSNG.

Data Availability Statement: The data presented in this study will be available on Github at <https://github.com/abbasmollaei/P4THLS> accessed on 15 November 2025.

Conflicts of Interest: The authors declare no conflicts of interest.

References

- Govindan, R.; Minei, I.; Kallahalla, M.; Koley, B.; Vahdat, A. Evolve or Die: High-Availability Design Principles Drawn from Googles Network Infrastructure. In Proceedings of the 2016 ACM SIGCOMM Conference, Florianopolis, Brazil, 22–26 August 2016; SIGCOMM '16; pp. 58–72. [CrossRef]
- Hauser, F.; Häberle, M.; Merling, D.; Lindner, S.; Gurevich, V.; Zeiger, F.; Frank, R.; Menth, M. A survey on data plane programming with P4: Fundamentals, advances, and applied research. *J. Netw. Comput. Appl.* **2023**, *212*, 103561. [CrossRef]
- Xu, Z.; Lu, Z.; Zhu, Z. Information-Sensitive In-Band Network Telemetry in P4-Based Programmable Data Plane. *IEEE/ACM Trans. Netw.* **2024**, *32*, 5081–5096. [CrossRef]
- Li, Y.; Ren, Z.; Li, W.; Liu, X.; Chen, K. Congestion Control for AI Workloads with Message-Level Signaling. In Proceedings of the 9th Asia-Pacific Workshop on Networking, Shanghai, China, 7–8 August 2025; APNET '25; pp. 59–65. [CrossRef]
- Turkovic, B.; Biswal, S.; Vijay, A.; Hüfner, A.; Kuipers, F. P4QoS: QoS-based Packet Processing with P4. In Proceedings of the IEEE International Conference on Network Softwarization (NetSoft), Tokyo, Japan, 28 June–2 July 2021; pp. 216–220. [CrossRef]
- Alcoz, A.G.; Vass, B.; Namyar, P.; Arzani, B.; Rétvári, G.; Vanbever, L. Everything matters in programmable packet scheduling. In Proceedings of the 22nd USENIX Symposium on Networked Systems Design and Implementation, Philadelphia, PA, USA, 28–30 April 2025; NSDI '25.
- AbbasMollaei, M.; Ould-Bachir, T.; Savaria, Y. Normal and Resilient Mode FPGA-based Access Gateway Function Through P4-generated RTL. In Proceedings of the 2024 20th International Conference on the Design of Reliable Communication Networks (DRCN), Montreal, QC, Canada, 6–9 May 2024; pp. 32–38. [CrossRef]
- Nickel, M.; Göhringer, D. A Survey on Architectures, Hardware Acceleration and Challenges for In-Network Computing. *ACM Trans. Reconfigurable Technol. Syst.* **2024**, *18*, 1–34. [CrossRef]
- Sozzo, E.D.; Conficconi, D.; Zeni, A.; Salaris, M.; Sciuto, D.; Santambrogio, M.D. Pushing the Level of Abstraction of Digital System Design: A Survey on How to Program FPGAs. *ACM Comput. Surv.* **2022**, *55*, 1–48. [CrossRef]
- Bosshart, P.; Daly, D.; Gibb, G.; Izzard, M.; McKeown, N.; Rexford, J.; Schlesinger, C.; Talayco, D.; Vahdat, A.; Varghese, G.; et al. P4: Programming Protocol-Independent Packet Processors. *SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 87–95.
- Cao, Z.; Su, H.; Yang, Q.; Shen, J.; Wen, M.; Zhang, C. P4 to FPGA-A Fast Approach for Generating Efficient Network Processors. *IEEE Access* **2020**, *8*, 23440–23456. [CrossRef]
- AbbasMollaei, M.; Ould-Bachir, T.; Savaria, Y. P4THLS: A Templated HLS Framework to Automate Efficient Mapping of P4 Data-Plane Applications to FPGAs. *IEEE Access* **2025**, *13*, 164829–164845. [CrossRef]
- Budiu, M.; Dodd, C. The P416 Programming Language. *ACM SIGOPS Oper. Syst. Rev.* **2017**, *51*, 5–14. [CrossRef]
- Goswami, B.; Kulkarni, M.; Paulose, J. A Survey on P4 Challenges in Software Defined Networks: P4 Programming. *IEEE Access* **2023**, *11*, 54373–54387. [CrossRef]
- Jasny, M.; Thstrup, L.; Ziegler, T.; Binnig, C. P4DB—The Case for In-Network OLTP. In Proceedings of the 2022 International Conference on Management of Data, Philadelphia, PA, USA, 12–17 June 2022; SIGMOD '22; pp. 1375–1389. [CrossRef]
- Castanheira, L.; Parizotto, R.; Schaeffer-Filho, A.E. FlowStalker: Comprehensive Traffic Flow Monitoring on the Data Plane using P4. In Proceedings of the IEEE International Conference on Communications (ICC), Shanghai, China, 20–24 May 2019; pp. 1–6. [CrossRef]
- Jin, X.; Li, X.; Zhang, H.; Soulé, R.; Lee, J.; Foster, N.; Kim, C.; Stoica, I. NetCache: Balancing Key-Value Stores with Fast In-Network Caching. In Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, 28 October 2017; SOSP '17; pp. 121–136. [CrossRef]
- Moro, D.; Sanvito, D.; Capone, A. FlowBlaze.p4: A library for quick prototyping of stateful SDN applications in P4. In Proceedings of the IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN), Leganes, Spain, 10–12 November 2020; pp. 95–99. [CrossRef]

19. Tulumello, A. P4 language extensions for stateful packet processing. In Proceedings of the International Conference on Network and Service Management (CNSM), Izmir, Turkey, 25–29 October 2021; pp. 98–103. [\[CrossRef\]](#)
20. Lu, R.; Guo, Z. An FPGA-Based High-Performance Stateful Packet Processing Method. *Micromachines* **2023**, *14*, 2074. [\[CrossRef\]](#)
21. Allard, F.; Ould-Bachir, T.; Savaria, Y. Enhancing P4 Syntax to Support Extended Finite State Machines as Native Stateful Objects. In Proceedings of the 2024 IEEE 10th International Conference on Network Softwarization (NetSoft), Saint Louis, MO, USA, 24–28 June 2024; pp. 326–330. [\[CrossRef\]](#)
22. Pontarelli, S.; Bifulco, R.; Bonola, M.; Cascone, C.; Spaziani, M.; Bruschi, V.; Sanvito, D.; Siracusano, G.; Capone, A.; Honda, M.; et al. FlowBlaze: Stateful Packet Processing in Hardware. In Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19), Boston, MA, USA, 26–28 February 2019; pp. 531–548.
23. Feng, W.; Gao, J.; Chen, X.; Antichi, G.; Basat, R.B.; Shao, M.M.; Zhang, Y.; Yu, M. F3: Fast and Flexible Network Telemetry with an FPGA coprocessor. *Proc. ACM Netw.* **2024**, *2*, 1–22. [\[CrossRef\]](#)
24. Chen, Y.; Layeghy, S.; Manocchio, L.D.; Portmann, M. P4-NIDS: High-Performance Network Monitoring and Intrusion Detection in P4. In *Proceedings of the Intelligent Computing*; Arai, K., Ed.; Springer: Berlin/Heidelberg, Germany, 2025; pp. 355–373. [\[CrossRef\]](#)
25. Xavier, B.M.; Silva Guimarães, R.; Comarela, G.; Martinello, M. MAP4: A Pragmatic Framework for In-Network Machine Learning Traffic Classification. *IEEE Trans. Netw. Serv. Manag.* **2022**, *19*, 4176–4188. [\[CrossRef\]](#)
26. Zeng, C.; Luo, L.; Zhang, T.; Wang, Z.; Li, L.; Han, W.; Chen, N.; Wan, L.; Liu, L.; Ding, Z.; et al. Tiara: A Scalable and Efficient Hardware Acceleration Architecture for Stateful Layer-4 Load Balancing. In Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI 22), Renton, WA, USA, 4–6 April 2022; pp. 1345–1358.
27. Han, Z.; Briasco-Stewart, A.; Zink, M.; Leeser, M. Extracting TCPIP Headers at High Speed for the Anonymized Network Traffic Graph Challenge. In Proceedings of the 2024 IEEE High Performance Extreme Computing Conference (HPEC), Wakefield, MA, USA, 23–27 September 2024; pp. 1–6. [\[CrossRef\]](#)
28. Heinanen, D.J.; Guerin, D.R. *A Single Rate Three Color Marker*; RFC 2697; RFC Editor: Fremont, CA, USA, 1999. [\[CrossRef\]](#)
29. Heinanen, D.J.; Guerin, D.R. *A Two Rate Three Color Marker*; RFC 2698; RFC Editor: Fremont, CA, USA, 1999. [\[CrossRef\]](#)
30. Laidig, R.; Dürr, F.; Rothermel, K.; Wildhagen, S.; Allgöwer, F. Dynamic Deterministic Quality of Service Model with Behavior-Adaptive Latency Bounds. In Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA), Niigata, Japan, 30 August–1 September 2023; pp. 127–136. [\[CrossRef\]](#)
31. Zhang, L.; Li, Y.; Yang, W.; Liu, Q.; Yang, R. PFTB: A Prediction-Based Fair Token Bucket Algorithm based on CRDT. In Proceedings of the International Conference on Computer Supported Cooperative Work in Design (CSCWD), Tianjin, China, 8–10 May 2024; pp. 928–934. [\[CrossRef\]](#)
32. Wang, S.Y.; Hu, H.W.; Lin, Y.B. Design and Implementation of TCP-Friendly Meters in P4 Switches. *IEEE/ACM Trans. Netw.* **2020**, *28*, 1885–1898. [\[CrossRef\]](#)
33. Liu, K.; Liu, C.; Wang, Q.; Li, Z.; Lu, L.; Wang, X.; Xiao, F.; Zhang, Y.; Dou, W.; Chen, G.; et al. An Anatomy of Token-Based Congestion Control. *IEEE Trans. Netw.* **2025**, *33*, 479–493. [\[CrossRef\]](#)
34. Kang, L.; Tian, L.; Hu, Y. A Programmable Packet Scheduling Method Based on Traffic Rate. In Proceedings of the 2024 IEEE 2nd International Conference on Control, Electronics and Computer Technology (ICCECT), Jilin, China, 26–28 April 2024; pp. 211–215. [\[CrossRef\]](#)
35. P4.org Applications Working Group. In-band Network Telemetry (INT) Dataplane Specification, ver. 2.1. Available online: https://p4.org/wp-content/uploads/sites/53/p4-spec/docs/INT_v2_1.pdf (accessed on 1 November 2025).
36. Tan, L.; Su, W.; Zhang, W.; Lv, J.; Zhang, Z.; Miao, J.; Liu, X.; Li, N. In-band Network Telemetry: A Survey. *Comput. Netw.* **2021**, *186*, 107763. [\[CrossRef\]](#)
37. Sardellitti, S.; Polverini, M.; Barbarossa, S.; Cianfrani, A.; Lorenzo, P.D.; Listanti, M. In Band Network Telemetry Overhead Reduction Based on Data Flows Sampling and Recovering. In Proceedings of the IEEE International Conference on Network Softwarization (NetSoft), Madrid, Spain, 19–23 June 2023; pp. 414–419. [\[CrossRef\]](#)
38. Zheng, Q.; Tang, S.; Chen, B.; Zhu, Z. Highly-Efficient and Adaptive Network Monitoring: When INT Meets Segment Routing. *IEEE Trans. Netw. Serv. Manag.* **2021**, *18*, 2587–2597. [\[CrossRef\]](#)
39. Zhang, Y.; Pan, T.; Zheng, Y.; Song, E.; Liu, J.; Huang, T.; Liu, Y. INT-Balance: In-Band Network-Wide Telemetry with Balanced Monitoring Path Planning. In Proceedings of the ICC 2023—IEEE International Conference on Communications, Rome, Italy, 28 May–1 June 2023; pp. 2351–2356. [\[CrossRef\]](#)
40. Bal, S.; Han, Z.; Handagala, S.; Cevik, M.; Zink, M.; Leeser, M. P4-Based In-Network Telemetry for FPGAs in the Open Cloud Testbed and FABRIC. In Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Vancouver, BC, Canada, 20 May 2024; pp. 1–6. [\[CrossRef\]](#)
41. Bhamare, D.; Kassler, A.; Vestin, J.; Khoshkholghi, M.A.; Taheri, J. IntOpt: In-Band Network Telemetry Optimization for NFV Service Chain Monitoring. In Proceedings of the ICC 2019—IEEE International Conference on Communications (ICC), Shanghai, China, 20–24 May 2019; pp. 1–7. [\[CrossRef\]](#)

42. Li, F.; Yuan, Q.; Pan, T.; Wang, X.; Cao, J. MTU-Adaptive In-Band Network-Wide Telemetry. *IEEE/ACM Trans. Netw.* **2024**, *32*, 2315–2330. [CrossRef]
43. Yu, M. Network telemetry: Towards a top-down approach. *SIGCOMM Comput. Commun. Rev.* **2019**, *49*, 11–17. [CrossRef]
44. Luo, L.; Chai, R.; Yuan, Q.; Li, J.; Mei, C. End-to-End Delay Minimization-Based Joint Rule Caching and Flow Forwarding Algorithm for SDN. *IEEE Access* **2020**, *8*, 145227–145241. [CrossRef]
45. Wei, B.; Wang, L.; Zhu, J.; Zhang, M.; Xing, L.; Wu, Q. Flow control oriented forwarding and caching in cache-enabled networks. *J. Netw. Comput. Appl.* **2021**, *196*, 103248. [CrossRef]
46. Chen, Y.F.; Lin, F.Y.S.; Hsu, S.Y.; Sun, T.L.; Huang, Y.; Hsiao, C.H. Adaptive Traffic Control: OpenFlow-Based Prioritization Strategies for Achieving High Quality of Service in Software-Defined Networking. *IEEE Trans. Netw. Serv. Manag.* **2025**, *22*, 2295–2310. [CrossRef]
47. Tiwana, P.S.; Singh, J. Enhancing Multimedia Forwarding in Software-Defined Networks: An Optimal Flow Mechanism Approach. In Proceedings of the International Conference on Advancement in Computation & Computer Technologies (InCACCT), Gharuan, India, 2–3 May 2024; pp. 888–893. [CrossRef]
48. P4 Language Consortium. *v1model.p4—Architecture for Simple_Switch*. Available online: <https://github.com/p4lang/p4c/blob/master/p4include/v1model.p4> (accessed on 1 October 2025).
49. AMD Vitis Networking P4 User Guide—Target Architecture (UG1308). 2023. Available online: <https://docs.xilinx.com/r/en-US/ug1308-vitis-p4-user-guide/Target-Architecture> (accessed on 1 October 2025).
50. Yang, X.; Sun, Z.; Li, J.; Yan, J.; Li, T.; Quan, W.; Xu, D.; Antichi, G. FAST: Enabling fast software/hardware prototype for network experimentation. In Proceedings of the International Symposium on Quality of Service, Phoenix, AZ, USA, 24–25 June 2019; IWQoS '19. [CrossRef]
51. Ibanez, S.; Brebner, G.; McKeown, N.; Zilberman, N. The P4->NetFPGA Workflow for Line-Rate Packet Processing. In Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Seaside, CA, USA, 24–26 February 2019; FPGA '19; pp. 1–9. [CrossRef]
52. P4 Language Consortium. *P4C: The P4 Compiler*. Available online: <https://github.com/p4lang/p4c> (accessed on 1 October 2025).
53. Cisco Systems. *TRex: A Stateful Traffic Generator*. Available online: <https://github.com/cisco-system-traffic-generator/trex-core> (accessed on 1 October 2025).
54. Abbasmollaei, M.; Ould-Bachir, T.; Savaria, Y. HLSCAM: Fine-Tuned HLS-Based Content Addressable Memory Implementation for Packet Processing on FPGA. *Electronics* **2025**, *14*, 1765. [CrossRef]

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.