Titre: Title:	Modular eCPRI Protocol Processing and Enhanced PTP-1588 Synchronization for 5G Fronthaul Using P4
Auteur: Author:	Atabak Nojavan
Date:	2024
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Nojavan, A. (2024). Modular eCPRI Protocol Processing and Enhanced PTP-1588 Synchronization for 5G Fronthaul Using P4 [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/62007/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Modular eCPRI Protocol Processing and Enhanced PTP-1588 Synchronization for 5G Fronthaul Using P4

ATABAK NOJAVAN
Département de génie électrique

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie électrique

Décembre 2024

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Modular eCPRI Protocol Processing and Enhanced PTP-1588 Synchronization for 5G Fronthaul Using P4

présenté par **Atabak NOJAVAN**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Jean-Pierre DAVID, président
Yvon SAVARIA, membre et directeur de recherche
François-Raymond BOYER, membre et codirecteur de recherche
Guy BOIS, membre

DEDICATION

To my family

ACKNOWLEDGEMENTS

First of all, I would like to express my sincere gratitude to my supervisor, professor Yvon Savaria, for his invaluable guidance, support, and encouragement throughout my research journey. Working with him has been a rewarding experience, and I have learned much from his expertise.

I am also deeply grateful to my co-supervisor, professor François-Raymond Boyer, for his continuous support and constructive feedback. His technical expertise has significantly contributed to the successful completion of this research.

I would also like to thank Bill Pontikakis for his contributions to this research. His hard work, dedication, and experience have been invaluable in the development of this project.

Finally, I would like to express my appreciation to the jury committee members, professor Jean-Pierre David and professor Guy Bois for reviewing this work. Their time and expertise are greatly valued.

RÉSUMÉ

L'émergence de la technologie de communication sans fil 5G a marqué le début d'une nouvelle ère d'innovation dans les télécommunications, exigeant des réseaux plus rapides et plus fiables. La réalisation de tels réseaux nécessite la mise en œuvre d'architectures réseau innovantes et plus flexibles.

L'interface commune publique radio améliorée (eCPRI) est une interface standardisée pour les réseaux d'accès radio ouverts (O-RAN). Elle joue un rôle crucial répondant aux nouvelles exigences de la 5G en prenant en charge les technologies de télécommunications fronthaul réseau modernes et en standardisant les protocoles fronthaul réseau, ce qui favorise la coopération inter-fournisseurs.

Le langage de programmation P4 est un outil puissant pour la réalisation d'architectures réseau flexibles. Ce langage de domaine spécifique (DSL) décrit le comportement de transfert de paquets dans les dispositifs réseau de plan de données tels que les commutateurs et les routeurs.

Ce travail explore l'application de P4 dans la mise en œuvre d'un pipeline de traitement de paquets eCPRI robuste et adaptable. Nous présentons des fonctionnalités complexes de traitement de paquets, notamment la concaténation, la déconcaténation et la synchronisation d'horloge. De plus, nous étudions les capacités et les limitations de P4 dans la mise en œuvre de fonctions de traitement de paquets complexes.

Étant donné l'évolution d'eCPRI et son potentiel d'amélioration future, une approche P4 flexible est essentielle. Pour notre application, P4 offre une modularité limitée. Pour relever ce défi, nous proposons un pipeline P4 modulaire intégrant un préprocesseur personnalisé pour permettre de futures extensions et la personnalisation de notre conception, ainsi que d'assurer sa compatibilité future.

Notre pipeline P4 intègre un module de synchronisation conforme au protocole de temps de précision IEEE-1588 (PTP). Pour rendre les commutateurs logiciels existants compatibles avec ce protocole, nous présentons un nouveau mécanisme de marquage temporel utilisant des horodatages standard et précis.

En utilisant notre implémentation d'un mécanisme de marquage temporel nouveau et plus précis pour le PTP-1588, nous avons réduit l'erreur de synchronisation de 24 000 microsecondes à 60 microsecondes (amélioration de 99,75 %).

Pour tester rigoureusement la fonctionnalité de notre unité de traitement de paquets eCPRI,

nous avons effectué des simulations approfondies à l'aide de commutateurs logiciels P4. Nous avons corrigé les limitations des commutateurs logiciels existants en apportant les modifications nécessaires pour permettre des tests précis des protocoles réseau sensibles au temps, tels que le PTP-1588.

Nous présentons également de nouveaux outils pour tester les protocoles réseau sensibles au temps sur les commutateurs P4, permettant aux chercheurs d'étudier les effets des erreurs d'horloge sur l'exécution des protocoles réseau sensibles au temps.

Grâce à nos recherches, nous avons acquis une connaissance approfondie des forces et des faiblesses de P4 pour le traitement complexe des paquets. Nos résultats contribuent à l'exploration continue des architectures de réseau programmables et de leur potentiel à révolutionner l'industrie des télécommunications. En étendant les capacités des commutateurs logiciels P4 existants, nous fournissons une plate-forme précieuse aux chercheurs pour effectuer des tests plus précis et complets des protocoles réseau sensibles au temps. Cela permet d'évaluer l'impact des erreurs d'horloge sur les performances du réseau et de développer des de réseau plus robustes et résilients.

ABSTRACT

The emergence of 5G wireless telecommunication technology has ushered in a new era of innovation in telecommunications. This demand for faster and more reliable network communications requires realizing more innovative and flexible network architectures. Enhanced Common Public Radio Interface (eCPRI), a standardized interface for Open Radio Access Networks (O-RANs), plays a crucial role in addressing these new demands by supporting modern network fronthaul telecommunications technologies and standardizing network fronthaul protocols, which fosters inter-vendor cooperation. Programming Protocol-Independent Packet Processors (P4) is a powerful tool for realizing flexible network architectures. This Domain-Specific Language (DSL) describes packet forwarding behavior in data-plane network devices such as switches and routers.

This work explores the application of P4 in implementing a robust and adaptable eCPRI packet processing pipeline. We present complex packet processing functionalities, including concatenation, de-concatenation, and clock synchronization. Furthermore, we investigate P4's capabilities and limitations in implementing complex packet processing functions.

Given the evolving nature of eCPRI and its potential for future improvements, a flexible P4-based approach is essential. For our application, P4 offers limited modularity. To address this challenge, we propose a modular P4 pipeline, incorporating a custom preprocessor to enable future extensions and customization of our design and ensure its forward compatibility. Our modular implementation reduces the number of lines of P4 code necessary for supporting future eCPRI message types.

Our proposed P4 pipeline incorporates a synchronization module adhering to the IEEE-1588 Precision Time Protocol (PTP). To make existing software switches compatible with such protocol, we present a new mechanism for timestamping using standard and precise timestamps. Using our implementation of the new and more precise time-stamping mechanism for the PTP-1588, we reduced synchronization error from 24,000 microseconds to 60 microseconds (99.75% improvement).

To rigorously test the functionality of our eCPRI packet processing unit, we conduct extensive simulations using P4 software switches. We address the limitations of existing software switches by making necessary modifications to enable accurate testing of time-sensitive network protocols, such as PTP-1588. We also present new tools for testing time-sensitive protocols on P4 switches, which allow researchers to study the effects of clock error on the execution of time-sensitive network protocols.

Through our research, we gain valuable knowledge of P4's strengths and limitations for complex packet processing. Our findings contribute to the ongoing exploration of programmable network architectures and their potential to revolutionize the telecommunications industry. By extending the capabilities of existing P4 software switches, we provide a valuable platform for researchers to conduct more precise and comprehensive testing of time-sensitive network protocols. This enables the evaluation of the impact of clock errors on network performance and the development of more robust and resilient network designs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

AP	Access Point
ASIC	Application-Specific Integrated Circuit
BBU	Base Band Unit
CMOS	Complementary Metal-Oxide-Semiconductor
C-RAN	Centralized Radio Access Network
CPRI	Common Public Radio Interface
CSR	Control and Status Registers
CU	Centralized Unit
D-RAN	Distributed Radio Access Network
DSCP	Differentiated Services Code Point
DSL	Domain-Specific Language
DU	Distributed Units
eCPRI	enhanced Common Public Radio Interface
eRE	eCPRI radio equipment
eREC	eCPRI radio equipment control
FPGA	Field Programmable Gate Array
INT	In-band Network Telemetry
MIMO	Multiple-Input Multiple-Output
MU	Mobile User
NGFI	Next Generation Fronthaul Interface
O-RAN	Open Radio Access Network
P4	Programming Protocol-independent Packet Processors
QoS	Quality of Service
RAN	Radio Access Network
RH	Radio Head
RRH	Remote Radio Head
RRU	Remote Radio Unit
RU	Radio Unit
SoC	System on a Chips

CHAPTER 1 INTRODUCTION

The proliferation of data-intensive applications, such as video streaming, virtual reality, and cloud computing, has increased the demand for higher bandwidth communications. As users expect to access and transmit large amounts of data seamlessly, network infrastructure must evolve to accommodate these growing requirements.

The rapid evolution of network technologies has necessitated innovative approaches to designing, implementing, and managing network infrastructure. Traditional hardware-based networking solutions, while reliable, often lack the flexibility and programmability required to adapt to emerging network paradigms.

1.1 Project overview

This project explores the potential of Programming Protocol-independent Packet Processors (P4) [1,2], a domain-specific language (DSL) for designing network devices, to bridge the gap between hardware and software, enabling greater customization and control over network behavior. Specifically, we utilize P4 to design and implement novel and modular packet processing pipelines for enhanced Common Public Radio Interface (eCPRI) [3]. P4's ability to describe and control network behavior at a granular level makes it well-suited for creating highly customized and efficient network devices. We created an enhanced and more flexible eCPRI packet processing P4 pipeline by taking inspiration from an existing Intel eCPRI IP core [4],

Using P4 to implement a packet processing pipeline instead of a proprietary circuit IP offers numerous benefits, such as increased flexibility and customization. While P4 excels at packet forwarding, it is not designed for packet manipulation or storage. Our eCPRI packet processing unit has complex functionalities for packet conversion and storage. Above standard network packet conversion, our pipeline can perform packet concatenation, packet de-concatenation and clock synchronization. This provides an ideal testbed for exploring the boundaries of P4's capabilities. By attempting to create such a pipeline using P4, we not only aim to develop a valuable network component but also gain valuable insights into P4's strengths and weaknesses.

As we will discuss in the next chapters, eCPRI is a new protocol with room for extension and customization. This necessitates a modular design that can be extended to support new eCPRI message types and custom functions for them. We used macro preprocessors to

incorporate modularity into our design and future-proof our implementation.

To facilitate comprehensive testing of our P4-based unit in a controlled environment, we modified existing software switches to enable extensive logic verification. We added tools for simulating clock errors in these switches.

This chapter will introduce fundamental concepts, protocols, and design principles that lay the groundwork for our work and provide in-depth discussions in the following chapters. In the next chapters, we will discuss the related research, the proposed eCPRI pipeline, the incorporation of modularity into our design, and our contribution to the existing software switches.

1.2 Evolving design of fronthaul networks and network equipment

The evolution of network fronthaul technologies has been crucial to mobile network development, especially with the transition from 4G to 5G. Fronthaul refers to the communication link between the baseband units (BBUs) and the remote radio heads (RRHs) or antennas in cellular networks. As network demands have increased, particularly with the rise of 5G, fronthaul technologies have had to evolve to support higher bandwidths, improved efficiency, and greater flexibility. This section outlines the progression from legacy proprietary interfaces to the standardized Common Public Radio Interface (CPRI), and finally to Enhanced CPRI (eCPRI), designed to meet the challenges of modern 5G deployments.

1.2.1 Legacy fronthaul architectures

Before the introduction of CPRI in 2003 [5], mobile network fronthaul architectures were largely proprietary, with each equipment vendor using custom-designed interfaces. These proprietary interfaces lacked interoperability, creating challenges for network operators wishing to integrate equipment from different vendors. Some early systems also used analog radio frequency (RF) transport through coaxial cables, which was susceptible to signal degradation, particularly at higher frequencies. As shown in 1.1, these systems relied on cell-site cabinets housing baseband and radio unit exclusive to each tower in a traditional Radio Access Network (RAN), also known as Distributed Radio Access Network (D-RAN).

This infrastructure-heavy approach led to high operational costs and scalability limitations, hindering the deployment of advanced mobile network technologies. These limitations highlighted the need for a more standardized and efficient approach to fronthaul, setting the stage for the development of CPRI.

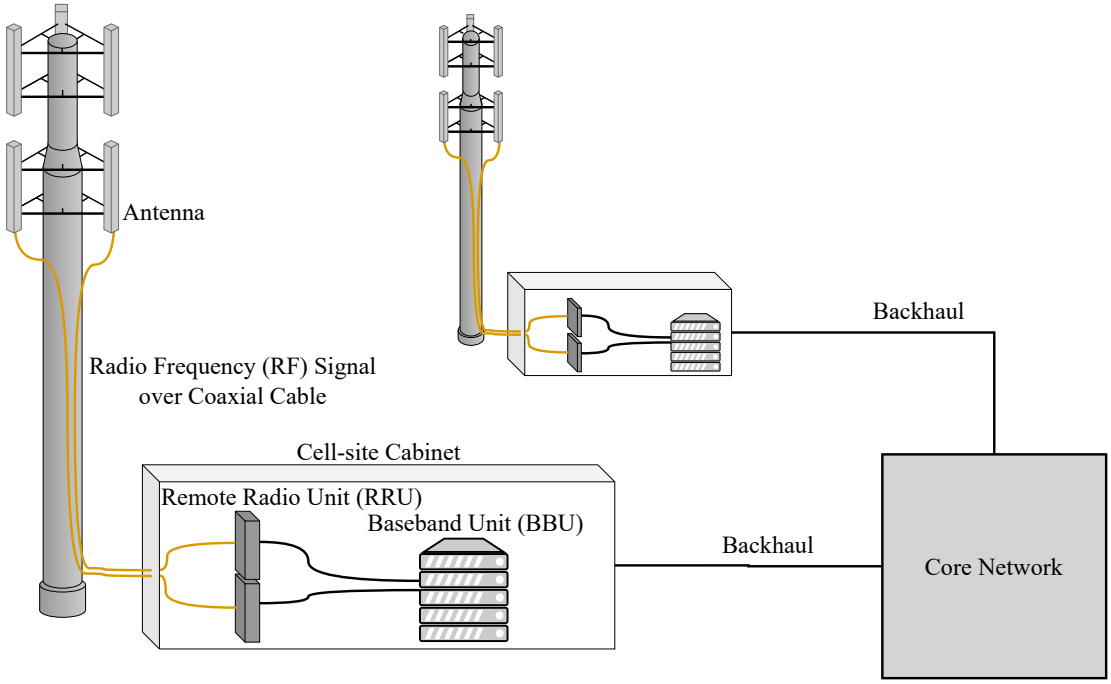


Figure 1.1 Legacy fronthaul network (D-RAN) connecting antenna to radio and baseband units placed near an antenna tower.

1.2.2 Common Public Radio Interface (CPRI)

To address the limitations of legacy architectures, the CPRI interface was introduced as a standardized solution for the fronthaul link. It was designed to create a standard interface specification that would enable the digital transport of RF signals over optical fiber, improving both signal quality and distance capabilities. Using fiber-optic cables allows the BBU be located several tens of kilometers from the antenna site [6]. By replacing traditional cable connections with fiber-optic cables, CPRI enabled RRH deployments, reducing infrastructure costs and simplifying network management. It also created a Centralized Radio Access Network (C-RAN) and introduced a new network segment called fronthaul [7].

This transition to a more flexible and efficient architecture marked a significant improvement in fronthaul architecture and paved the way for developing more advanced cellular networks, including 5G.

Although CPRI aimed to foster interoperability between equipment from different vendors, full compatibility was not always achieved in practice [8]; however, CPRI's ability to support new radio technologies and network architectures made it a significant step toward that goal. Nevertheless, with the advent of 5G, CPRI's limitations—such as its high bandwidth requirements, utilization of fixed frame structure [9], and lack of scalability became apparent

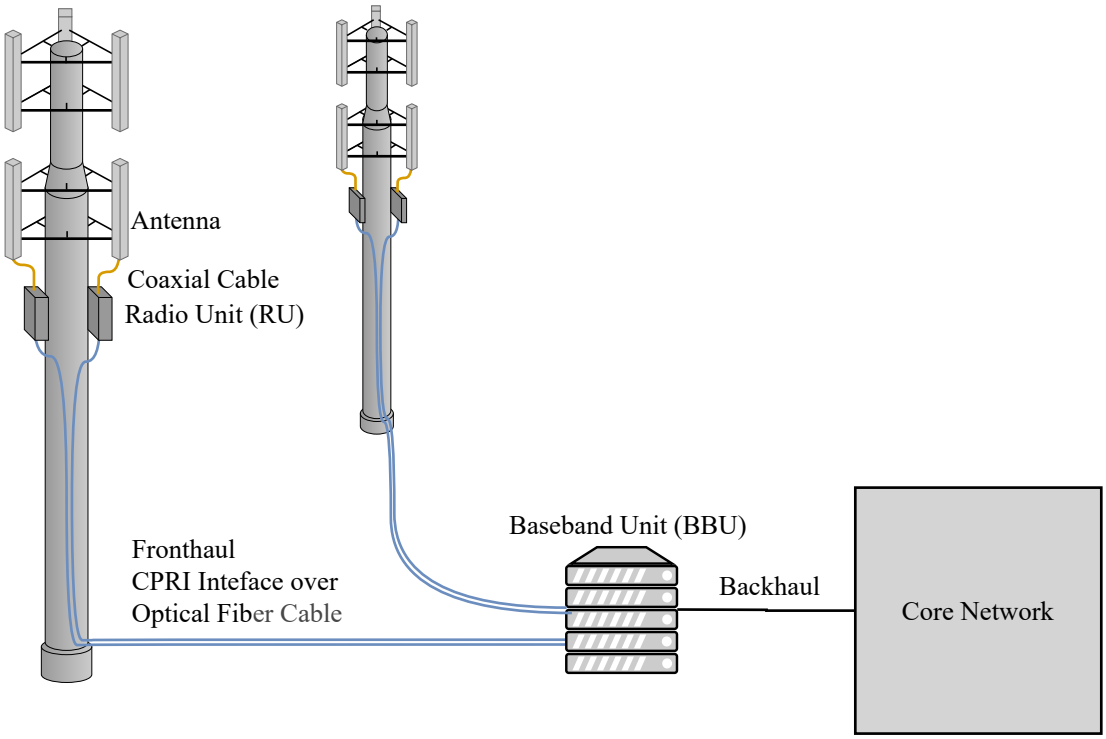


Figure 1.2 CPRI connecting baseband unit and radio unit in a C-RAN architecture.

[7], necessitating further innovations in fronthaul technologies.

1.2.3 Enhanced Common Public Radio Interface (eCPRI)

To address the challenges posed by 5G networks, eCPRI was introduced in 2017 as an evolution of the CPRI standard [9] intended for future O-RAN (Open Radio Access Network). Unlike CPRI, which used time-division multiplexing, eCPRI adopted a packet-based transport system (Ethernet/IP) [3], significantly improving efficiency. This new approach allowed better network planning and resource utilization. Moreover, eCPRI is more flexible than CPRI, allowing for dynamic resource allocation and a wider range of functional splits between the BBU and RU [6]. Additionally, with the emergence of O-RAN (Open Radio Access Network) standards, multi-vendor interoperability became possible, further driving innovation and reducing costs in deploying 5G networks.

Figure 1.3 displays the application of a eCPRI interface in fronthaul of a O-RAN. In a common version of these architecture, each tower is connected to a network of Distributed Units (DUs) and a Centralized Unit (CU), instead of a single exclusive one BBU. These units are connected using midhaul network. Implementing such architecture improves resource utiliza-

tion and efficiency. Moreover, since the eCPRI is an open standard, each node or interface in this architecture can be manufactured and operated by a different entity. Expanding such architecture is much easier than older generations since DU units added for each tower will contribute to the throughput of larger O-RAN.

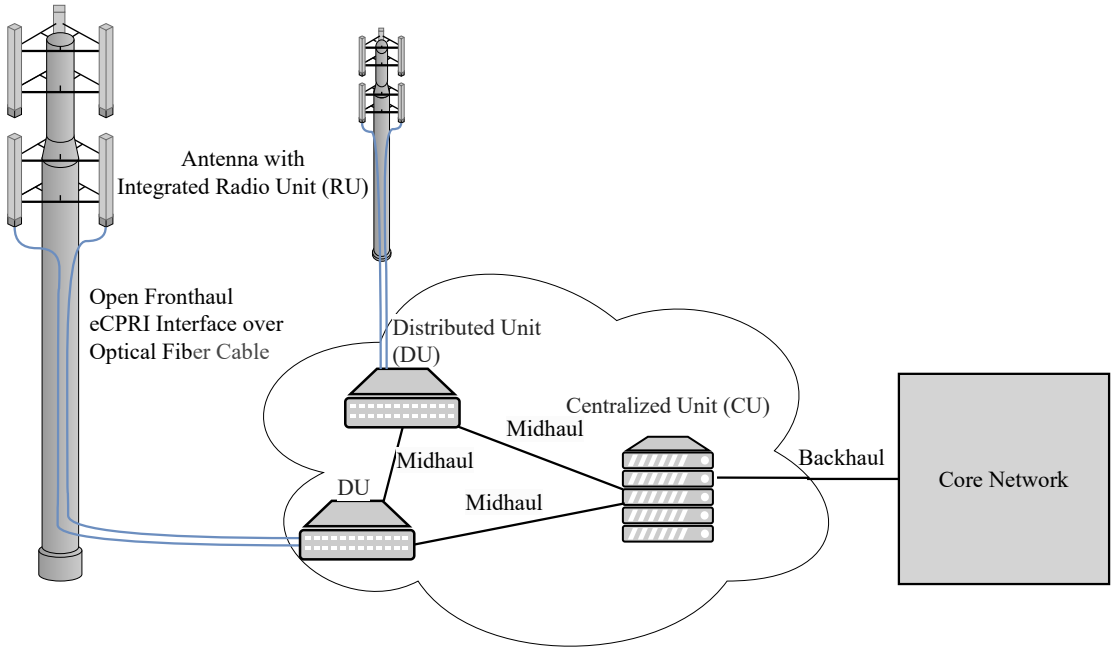


Figure 1.3 eCPRI connecting radio units to open network of distributed and centralized units in an O-RAN architecture.

Alongside eCPRI, the Next Generation Fronthaul Interface (NGFI) [10] from the IEEE 1914 working group is another standard for encapsulating fronthaul traffic inside Ethernet frames.

1.3 P4 language

P4 [1, 2] is a domain-specific programming language designed explicitly for expressing the behavior of network devices, such as switches and routers [11]. It offers a high-level abstraction for designing packet processing pipelines, allowing network engineers and researchers to define custom forwarding logic without delving into the complexities of hardware-specific programming languages.

P4 models the packet processing pipeline at its core as a series of stages, including ingress, parser, control plane, deparser, and egress. Each stage can perform specific operations on the packet, such as matching against headers, modifying fields, and making forwarding decisions. A key concept in P4 is the match-action paradigm, where packets are matched against a

set of rules, and corresponding actions are taken. This enables flexible and efficient packet processing. Even though P4 is primarily a language for describing forwarding behavior, it is flexible enough to describe packet conversion. Removing and adding headers are the primary tools that P4 provides us for this task.

Its high level of abstraction makes a flexible design choice. P4 enables rapid prototyping and experimentation with different network protocols and algorithms, facilitating innovation and customization. Designers can implement and verify the design in a software environment before final hardware implementation and avoid costly mistakes. Moreover, P4 hardware switches can be the most efficient candidates for networks with fast-evolving requirements and protocols, such as radio access networks in 5G telecommunications.

One of P4's significant advantages is its hardware independence and portability. P4-based designs can be easily adapted to different hardware platforms, reducing the time and cost associated with porting network applications.

Moreover, P4 is extensible, allowing for the creation of custom data plane operations and control plane interfaces using extern objects. These externs can be implemented using other programming languages or hardware description languages. This enables network engineers to develop tailored solutions for specific use cases that are not easy to implement using P4.

P4 has gained significant traction in the networking community and is used in research and commercial projects. Its ability to provide a flexible and powerful way to program network devices has made it a valuable tool for network engineers and researchers.

1.3.1 P4 limitations and externs as a solution for them

P4 is designed to be a relatively minimalist language, focusing on essential constructs for packet processing. While P4 has a core language of fewer than 40 keywords, it can be extended with externs to access advanced networking functionalities not directly supported by the core language. Externs allow P4 programs to interact with external components or systems, such as hardware accelerators, control planes, or other entities. This enables P4 to leverage the capabilities of these external components and create more complex and powerful network devices. By offloading complex operations or functionalities to external components, externs enable the creation of implementations that would be difficult or inefficient to achieve solely within the P4 language. This flexibility allows P4 to be used for a broader range of network applications and to tackle more challenging tasks.

For example, while P4 is efficient at manipulating packet headers and controlling forwarding behavior, it may not be as well-suited for tasks like checksum calculation, encryption, packet

concatenation, and de-concatenation. In such cases, externs can be used to offload these computationally intensive operations to external hardware or software implementations. Externs can be defined as functions or objects and implemented in various programming or hardware description languages.

By utilizing externs, P4 programs can benefit from the efficiency and specialized capabilities of external components, while maintaining a lean and focused core language. This modular approach allows for greater flexibility and customization in network device design.

Moreover, Externs in P4 are constructs for extending the language with architecture-specific functionalities implemented outside the P4 program for specific P4 hardware. For example, some specific hardware platforms for P4 language pipelines might provide their own functions, such as checksums or memory units. These functions can be accessed by using externs provided by the manufacturer.

Externs can be functions or objects with methods, similar to classes in object-oriented programming. These externs can be implemented in software environment using programming languages such as C++ in a software scenario. On the other hand, they can be implemented in hardware using hardware description languages. Moreover, high-level synthesis can implement externs and hardware environments directly from programming languages like C++.

Externs also interface the P4 pipeline and underlying hardware or software implementations. This interface can be used to control the P4 pipeline from other hardware sections or directly connect two parallel P4 pipelines together. Common functionalities can be reused across different P4 programs.

1.4 Software switches and their role in verification of P4 pipelines

Errors can occur during the design phase of a packet processing pipeline, causing issues such as packet loss or the creation of malformed packets in the final implementation. These errors can be overlooked if they occur only in specific scenarios and in response to specific inputs. Comprehensive testing is crucial to prevent such issues.

Software switches play a crucial role in developing and verifying a packet processing pipeline. Even if the final platform of a designed packet processing pipeline is hardware, designers first need to test the logic functionality of their pipeline in a software environment. By doing so, they can streamline the design process and prevent the lengthy and expensive debugging operation of their hardware implementations.

The simple switch behavioral model, known as simple_switch, is a crucial component of the P4 programming ecosystem. Developed as part of the BMv2 (Behavioral Model version

2) [12] framework by P4 org [11], it is a software implementation of a network switch that acts as a reference target for P4 programs. This model allows developers to compile and run their P4 code on a software switch, providing a flexible and accessible platform for development and testing before deploying to hardware.

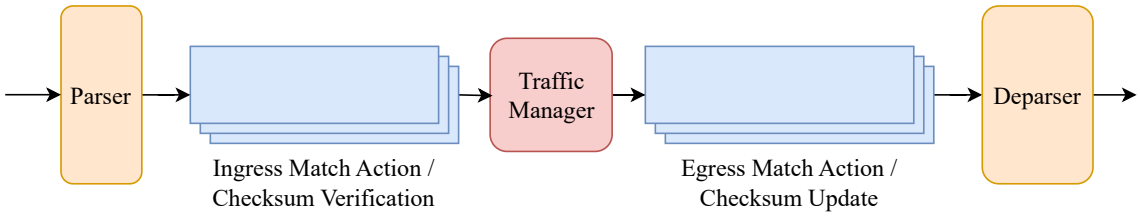


Figure 1.4 Schematic of V1 model

Simple_switch implements the V1Model architecture, which closely aligns with the abstract switch model described in the P4 specification. This architecture consists of seven programmable components: a parser, a checksum verification control block, ingress and egress match-action processing control blocks, a checksum update control block, a deparser, and a traffic manager. This structure enables P4 developers to define complex packet processing behaviors within a familiar and standardized framework. Figure 1.4 displays a schematic of this model combining checksum units and match-action processing control blocks. P4 programs written for this model can be compiled into a JSON file using the P4C [13] compiler, which is then used to configure the BMv2 software switch. This workflow facilitates rapid prototyping and testing of P4 programs in a software environment, making it an invaluable tool for P4 developers.

1.5 Synchronization requirements of fronthaul networks

Synchronization is a cornerstone of digital communication networks, ensuring the integrity and continuity of data transmission. Accurate clock synchronization is essential to prevent errors and information loss [14]. By aligning clocks across the network, we can maintain consistent timing relationships, enabling devices to interpret and process data correctly.

In particular, precise time synchronization is even more crucial in modern cellular technologies like 4G and 5G. Base stations and backhaul equipment must maintain accurate timing in these networks to enable coordinated transmissions and prevent interference [15]. This is particularly critical in techniques like Multiple-Input Multiple-Output (MIMO), where signals from multiple antennas or base stations must arrive in phase for optimal performance. If time synchronization is not maintained, these signals can become misaligned, leading to destructive

interference, degraded data throughput, and potential service disruptions.

One example of these new technologies with strict synchronization requirements is Carrier Aggregation (CA), a technique used in wireless communication to increase the data rate per user. It involves assigning multiple frequency blocks (subcarriers) to the same user, effectively increasing the bandwidth available for data transmission. Inaccurate synchronization disrupts the orthogonality of subcarriers, leading to Inter-Carrier-Interference (ICI) and a significant decline in system performance [16].

In large-scale deployments like C-RANs, where multiple RUs are coordinated by centralized BBUs, lack of synchronization can cause instability across the entire network [17]. Tasks like frame alignment and coordination of transmission timing between multiple RUs become difficult, resulting in performance degradation, increased packet loss, and inefficient spectrum usage. Without proper synchronization, the network may fail to smoothly transfer ongoing calls or data sessions from one cell to another, leading to dropped calls and interrupted data services [18]. This will be especially problematic in dense urban environments where handovers occur frequently as users move between small cells while walking or driving.

Coordinated Multi-Point (CoMP), a candidate technology for 5G, is another cellular network technology that improves performance by coordinating the transmission and reception of data from multiple base stations [19]. This approach allows for more efficient use of network resources and can enhance coverage, capacity, and user experience. Precise cell synchronization is crucial to prevent interference between downlink and uplink transmissions and between simultaneous transmissions with multiple RUs. By ensuring that downlink and uplink timeslots do not overlap, synchronization eliminates a significant source of interference and improves overall network performance.

Moreover, fifth-generation mobile networks provide a wide range of location-based services, such as location-based data streaming [20]. Precise positioning techniques for advanced location-based services in 5G networks have been developed. Many of these techniques rely on the cooperation between Access Points (APs) and Mobile Users (MUs). By leveraging the capabilities of APs and MU devices, these positioning methods can provide accurate location information for various applications [21]. Network and application layer services can access location data from a cloud-based location database. This enables the development of intelligent location-based data access streaming services. These cooperative positioning techniques in 5G networks require precise synchronization between APs and Mobile Users MUs. [17, 22]

1.6 Contributions

During this project we:

1. Proposed a complete eCPRI over Ethernet packet conversion unit implemented using P4 language. Our P4 pipeline comprises many advanced subsystems such as payload extraction, concatenation, de-concatenation, and PTP-1588 synchronization units. This proposed unit will perform only on the data plane and will be totally independent from the control plane.
2. Incorporated a novel modularity approach to our design, ensuring forward compatibility and extensibility.
3. Studied different approaches to packet concatenation, a vital network functionality, and acknowledged opportunities and disadvantages of each approach.
4. Studied P4 language shortcomings in area of payload extraction and processing
5. Extended existing "simple_switch" in BMv2 [12] environment to support Unix type precise timestamps used in our PTP-1588 implementation.
6. Added tools to BMv2 for testing and analyzing clock difference between two switches in software environment.
7. Found and fixed several software bugs in BMv2.

1.7 Work outline

This work is a dissertation by article and the remaining of it is organized as follows:

Chapter 2 presents the state of the art and theoretical background of our project.

In chapter 3, we will present the addition of standard timestamps to a standard software switch, fix its bugs, and extend it with advanced testing tools.

Chapter 4 is an article in which we discuss the details of implementing an eCPRI over Ethernet conversion unit using the P4 language and explain the implementation details of each packet processing subsystem. We also discuss the details of our modularity framework.

CHAPTER 2 LITERATURE REVIEW

This literature review chapter comprehensively overviews existing network packet processing technologies research. It focuses mainly on the subsystems we implemented for our eCPRI Packet processing pipeline and the role of the P4 language in implementing them. By examining the relevant research, we aim to establish the context for our research and identify the knowledge gaps our work seeks to address. This literature review will serve as a foundation for understanding this field's current state and our research's potential contributions.

Despite the importance of programmable switches and P4 language, there is a notable dearth of research specifically addressing substituting hardwired systems with P4 code. While some work has implemented eCPRI processing units [4, 23–26], no study has approached this challenge using P4. This can be attributed to the relatively recent emergence of P4 as a programming language and its ongoing evolution. The same issue applies to the subsystems necessary for the realization of our proposed P4 pipeline, and many of the subsystems implemented in our proposed pipeline are not implemented using conventional P4 methods.

Some research has approached the issue of variable-length packet payload extraction in P4 [27–31]. Most of these implementations have applications in deep packet inspection. Some other works experimented with the concept of payload feature extraction with similar applications. Time synchronization was implemented using P4 [32] but not in BMv2 software switch. Some research was conducted on P4 language modularity [33] or reduction of its voluminosity [34]. Most research on the issue of modularity aims to add completely different and independent packet processing algorithms to an existing pipeline. In this work, however, we aim to achieve modularity for different data and header types rather than different algorithms.

2.1 Precision Time Protocol (PTP) background

The Precision Time Protocol (PTP), specified under the IEEE-1588 standard [35], is a protocol designed for clock synchronization and one-way delay measurement throughout computer Local Area Networks (LANs), where it can provide synchronization with sub-microsecond accuracy.

PTP-1588 messages can be carried in Ethernet frames in the network's data link layer. The EtherType field value of "0x88F7" is assigned to the precision time protocol over IEEE 802.3 Ethernet [35].

2.1.1 PTP exchange and timestamp requirements

PTP-1588 calculates clock offsets by measuring timestamps at primary and subordinate clocks.

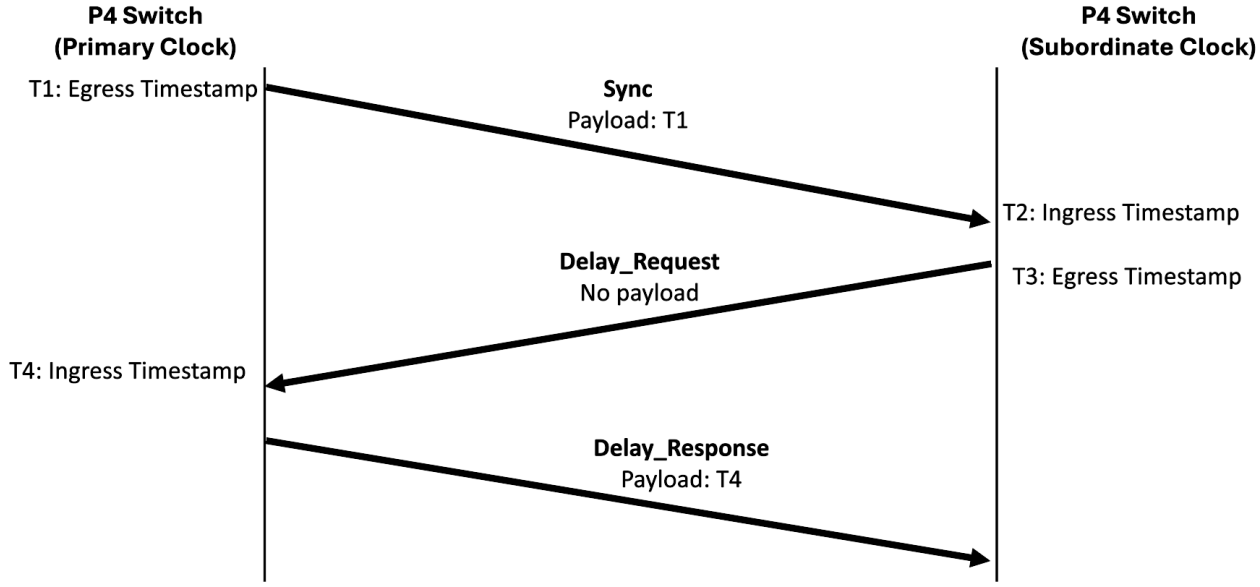


Figure 2.1 3-messages PTP-1588 Exchange

To do so, ingress and egress timestamps from the primary clock and two similar timestamps from the subordinate clock are needed. The primary clock is assumed to be highly accurate, and subordinate clocks adjust their time by measuring and compensating for the time difference relative to the primary clock. As shown in figures 2.1 and 2.2, the four necessary timestamps can be exchanged using either a 3-message or a 4-message PTP exchange. After complete packet processing, the 3-message exchange is suitable for switches that can add timestamps to PTP messages. Otherwise, the egress timestamp has to be recorded and sent to the subordinate switch using a follow-up message. 3-message exchange reduces network traffic, and if it is implemented in P4 no cloning operation will be required.

By utilizing simple calculations involving timestamp measurements, the clock offset between two switches can be determined:

$$\begin{aligned} T2 - T1 &= delay_{S1->S2} + \text{clock\_offset}. \\ T4 - T3 &= delay_{S2->S1} - \text{clock\_offset}. \end{aligned} \tag{2.1}$$

This protocol works best in networks with stable and symmetrical delays between the primary

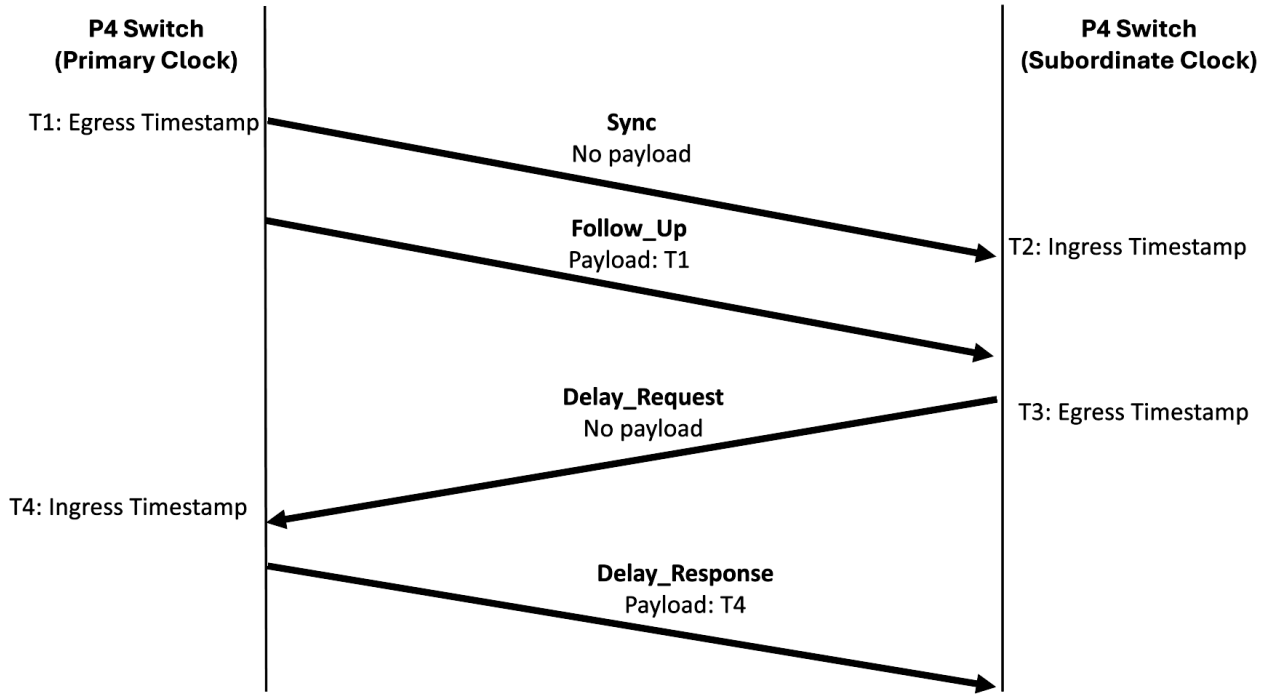


Figure 2.2 4-message PTP-1588 Exchange

and subordinate clocks. Assuming the delay between the two switches is symmetrical, we can derive equation (2.2) from equation (2.1).

$$\frac{(T2 - T1) - (T4 - T3)}{2} = \text{clock\_offset}. \tag{2.2}$$

PTP-1588 uses 80-bit timestamp fields, where the 64 least significant bits are the actual timestamp. These 64 bits are divided into two equal sections, where half represent seconds and the other measures fractions of a second in nanoseconds [36]. These timestamps should also measure time in seconds since the Unix Epoch (starting January 1, 1970, at 00:00:00 UTC). Implementing PTP-1588 on any system requires access to such standard timestamps.

2.2 Non-P4 implementations of eCPRI over Ethernet packet processing units

While efforts have been made to implement eCPRI over Ethernet, limited exploration has been made into leveraging P4 to achieve this. In this section, we will review some of these non-P4 approaches.

2.2.1 Intel eCPRI IP core

We took inspiration from the block diagram of Intel IP core [4]. This versatile unit incorporates a suite of subsystems for packet processing, including classification, header manipulation, and advanced concatenation/de-concatenation capabilities. The Intel FPGA IP core, which adheres to the eCPRI specification version 2.0, is intended to be paired with custom client logic within FPGA-based smart-NICs. The client logic generates raw messages, which the Intel IP core processes to create Ethernet frames containing eCPRI packets. The Intel IP core also handles packet concatenation and de-concatenation for specific eCPRI message types. The client logic attached to this IP transmits the main message, such as IQ_data, the main payload of eCPRI message type 0, through the primary input channel. In contrast, the eCPRI message type number and header field values, such as pc_id and seq_id, are conveyed via sideband signals. Moreover, Intel IP will use values in control and status registers (CSR) to populate Ethernet header fields. The client logic also controls this register. Moreover, these streams will go through the eCPRI IWF 0 unit, facilitating Intel IP's backward compatibility with older CPRI protocols. If the option is enabled, Intel IP will concatenate the data.

By converting Ethernet frames containing eCPRI messages into parallel data streams, the Intel eCPRI IP core creates eCPRI interfaces in FPGA-based 5G fronthaul network designs. These eCPRI interfaces connect eCPRI radio equipment control (eREC) to eCPRI radio equipment (eRE) via a fronthaul transport network.

In addition to Intel's proprietary eCPRI FPGA IP core [4], other alternative hardware implementations of eCPRI packet processing units have not relied on P4.

2.2.2 Using network IP cores for hardware implementation

Another study [24] utilized an Ethernet FPGA IP core to implement an eCPRI over an Ethernet processing unit. The study used FUSION IP cores to achieve low latency for both CPRI and eCPRI fronthaul traffic. This implementation offers a more adaptable solution using a general-purpose Ethernet IP core than Intel's IP core. The solution delivers a complete Ethernet fronthaul ecosystem, with all components required to implement the transport network for advanced 5G telecommunications applications such as eCPRI framer and deframer. The final hardware implementation of this work achieves an end-to-end maximum delay of 142 up to 721 nanoseconds.

While approaches in [4, 24] offer hardware-accelerated solutions, they rely on proprietary IP cores. In contrast, P4, as an open-source language, provides a more flexible and adaptable

approach to network programmability, enabling easier customization and evolution as eCPRI standards advance.

2.2.3 Implementing eCPRI processing unit using application-specific integrated circuit

In a different hardware implementation of eCPRI on Ethernet [25, 26], the researchers implemented an Application-Specific Integrated Circuit (ASIC) supporting eCPRI for 5G/LTE hybrid networks. They demonstrated the performance of their 16 nm Complementary Metal-Oxide-Semiconductor(CMOS) ASIC design over 20km of optical fiber, achieving 200Gb/s and 400Gb/s transmission rates.

Although such a design is a very efficient implementation, the ever-changing nature of the next generation of fronthaul networks will soon require a more flexible design. Such a design might fail to attract network vendors, as modern telecommunications networks' eCPRI protocol and O-RAN are evolving. The P4 language can provide a more future-proof solution.

2.2.4 Hardware-software implementation of eCPRI processing unit

One study [23] implemented an eCPRI packet processing module using a Zynq FPGA board, integrated with a PetaLinux-based Linux distribution. PetaLinux [37] is a software development kit for embedded Linux systems running on AMD (former Xilinx) devices. It provides an environment for building and deploying Linux images on AMD System on Chips (SoCs) and Field Programmable Gate Arrays (FPGAs). It also utilizes two IP cores, an Encapsulator IP that generates Ethernet frames by adding headers to the received data, and a parser IP for categorizing Ethernet frames. This implementation offers better flexibility, but the traffic rate is limited to 10Gb/s. Using proprietary and fixed IPs is another issue in this implementation.

2.3 P4 utilized for general processing of eCPRI packets

No work has used P4 for eCPRI over Ethernet conversion, but it was used for the general forwarding of eCPRI packets.

One work [38], proposes a novel fronthaul slicing architecture to improve the efficiency and flexibility of 5G networks using P4 language. Network slicing divides a network into multiple logical networks, each with their own set of resources and Quality of service parameters. By slicing the fronthaul, network operators can allocate resources to different services based on

their requirements, improving network efficiency and reducing costs. Researchers in the work mentioned above used P4 language to implement a packet categorization switch on an Intel Tofino-based switch.

Their proposed P4 pipeline detects the type of packet, whether intended for radio equipment or for radio equipment control, and will forward the packet to the correct destination. Their implementation was able to update routing information entirely in the data plane with low latency (0.5 μs - 2 μs) with no extra information needed from the control plane.

Another work [39] explored network slicing using P4 and implemented it on P4-NetFPGA framework. The proposed work has been validated in an actual 5G infrastructure. This uses data such as 5G user source/destination IPs, 5G user source/destination ports, and Differentiated Services Code Point (DSCP) to reroute the 5G network packets. DSCP is a mechanism for classifying network traffic that ensures Quality of Service (QoS). The work was validated in real 5G infrastructure.

These studies display P4's capability in describing and creating pipelines fast and efficiently enough for use in the next generation of mobile telecommunications.

2.4 Extracting variable-length packet payload

Variable-length packet payload extraction is an essential subsystem for concatenation and de-concatenation in our proposed P4 pipeline. It is a crucial step in realizing a concatenation unit. This section reviews works involving variable-length payload or header extraction and processing.

2.4.1 Total payload extraction

Other works have studied payload extraction, mainly used in deep packet inspection [27–31]. Researchers in these studies tried implementing variable-length packet payload extraction using constant-length headers and recirculation. They perform repetitive recirculation and parsing using a constant-size header to extract a specific field inside the packet with a variable size, such as URLs. The proposed algorithm will repeat this process until the entire URL is extracted. This technique is widely used for detecting malicious URLs inside a packet as part of a firewall implemented at the network or transport layer. Figure 2.3 displays a generic example of a deep packet inspection mechanism using constant-length headers. Finally, the data is stored in P4 registers for further processing. This system reads and categorizes the domain names in the data layer without using the control plane [31].

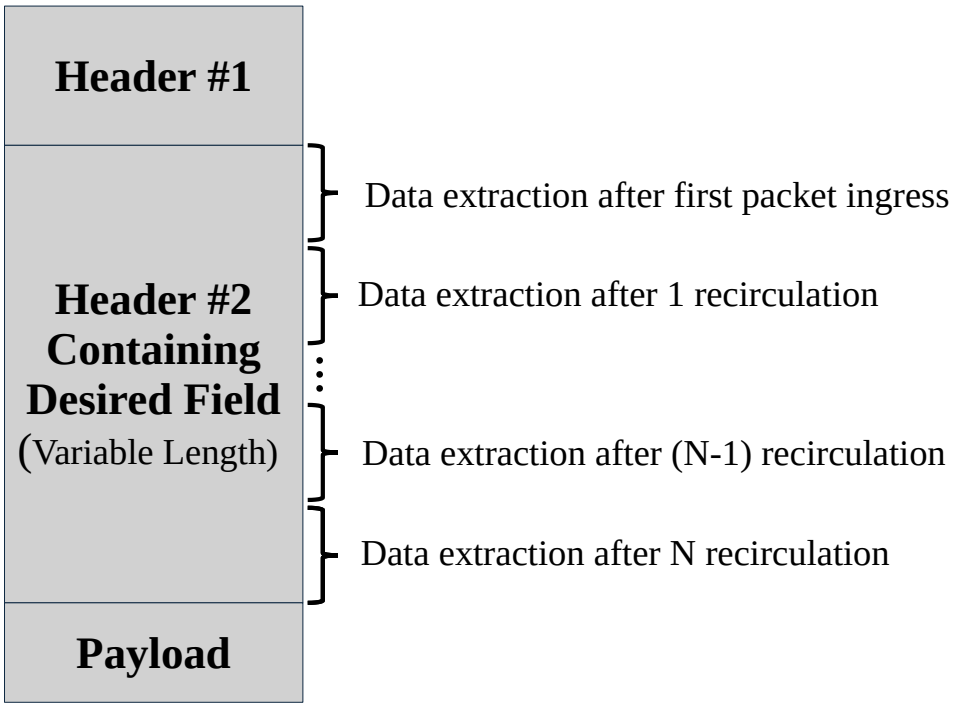


Figure 2.3 Schematic of deep packet inspection using consecutive recirculation and constant size header extraction

Using recirculation for parsing introduces latency proportional to the length of the domain name. Each recirculation cycle adds additional processing time, potentially impacting the overall packet processing performance and making the final implementation unsuitable for use in 5G network fronthaul with strict delay requirements. This technique is implemented due to limitations in the maximum size of headers in the utilized platforms. The envisioned target of our eCPRI processing pipeline is an FPGA switch, which also contains logic for eCPRI raw message generation. Therefore, such a limitation in header size will not be an issue for the final eCPRI processing unit.

2.4.2 Payload feature extraction

Another option for deep packet inspection is to forgo total payload extraction and only extract a section of the payload.

One work [40], recognizes the complexity of extracting the entire packet as it is and instead chooses to follow the other approach. Instead of using inefficient recirculation for extracting the packet payload, they propose extracting features of the packet in the data plane using a bloom filter and transferring this information more efficiently to the control plane. In the control plane, these features are used to train a machine-learning classifier. This trained

model will be used to detect malicious activity and possible network intrusions.

This study recognizes the value of payload extraction and data processing in the network data plane. However, it chooses not to perform total packet payload extraction because the extracted payload data needs to be transferred to the control plane through registers with two-way access. Such a transfer can result in a severe data rate penalty. This issue will not apply to our proposed eCPRI conversion unit, as we aim to perform all the operations in the data plane with no control plane involvement.

2.5 Storing variable-length packet payload

Packet payloads must be stored and combined into larger frames to implement packet concatenation. While concatenation is crucial to many protocols and networks, the P4 language does not offer a straightforward implementation approach.

Limited research, such as [41, 42], has addressed the storage of variable-length packet payloads. Most studies focus on small packet sizes with specific applications for the Internet of Things (IoT) packet aggregation. Implemented on a Tofino chip, a header with a different size is defined for every possible payload length. Such an approach offers limited expandability, its P4 code is verbose, and future resulting FPGA implementations will be inefficient.

Researchers in these studies have explored using multiple one-dimensional registers to create stacks of arrays resembling a large two-dimensional array for packet payload storage. As discussed in chapter 3, P4's lack of native support for two-dimensional registers restricts iteration to one dimension at a time, requiring compile-time-defined values to access elements in the other dimension. This issue will limit the maximum size and expandability of the memory. This P4 limitation will only exacerbate the challenges when the unit performs concatenation of multiple frames of the same type, effectively requiring a 3-dimensional register array. This makes the approach in [41, 42] unsuitable for our case. An external memory was created using an external object to store the extracted packet payload, creating an expandable implementation with improved flexibility. Connecting such memory to the P4 pipeline will require careful study of P4 limitations and eCPRI requirements.

2.6 Time synchronization unit

Implementing PTP-1588 in the data plane using P4 is relatively straightforward and has been done in some research [32]. This work implemented the data plane PTP on a Barefoot Tofino switch using the P4 programming language. When testing their implementation between two

hardware switches with four hops between them, they achieved a median synchronization of 19 nanoseconds and a 99th percentile synchronization error of 47 ns.

The availability of high-precision timestamps with the correct format is crucial for successful PTP implementation. As discussed in chapter 4, the BMv2 [12] software switch suffers from address format mismatch and lack of precision in timestamps. We added new timestamps, including an automatic egress timestamp and an option to add simulated clock errors. These additions allowed us to implement and rigorously test a 3-message PTP in the BMv2 software environment.

2.7 In-band network telemetry

Another paper [43] presents a novel approach to In-band Network Telemetry (INT) using P4 to program FPGA data planes and using the cycle counter of the FPGA as the clock. The proposed INT approach leverages the FPGA's precise clock for accurate timestamping. INT is a promising approach to network monitoring that embeds telemetry data directly within network traffic, enabling real-time insights into network performance, quality of service, and security.

This implementation shares similarities with our proposed PTP-1588 solution, as both involve precise timing measurements and executing calculations on timestamps in P4. This demonstrates the feasibility of implementing PTP-1588 on FPGAs.

2.8 P4 modularity and extendability

P4 programs often lack modularity and portability, being tightly coupled to specific hardware architectures. Additionally, P4 follows a heterogeneous programming model, meaning the different parts follow different models. For example, while P4 parsers are finite state machines, control blocks follow the imperative programming model [33].

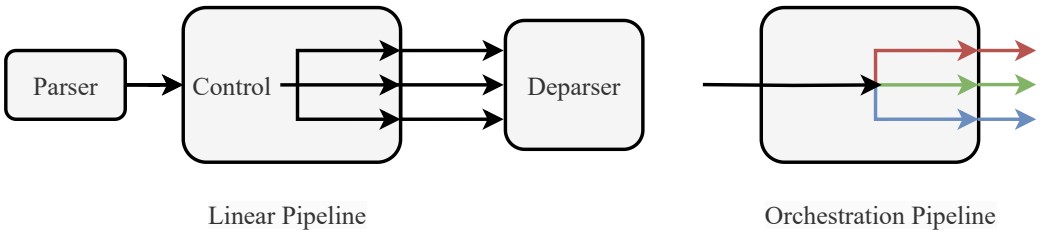


Figure 2.4 Linear and orchestration pipelines.

One work [33] proposed an extended P4-based programming language named µP4 that can be compiled to the generic P4. This approach will allow them to program on a higher abstraction level with better modularity while using the existing P4 infrastructure, such as P4C [13], the standard P4 compiler. Their proposed language uses two different pipelines:

1. A linear pipeline with 3 stages: parser, control, and deparser processing packets consecutively. This pipeline can produce multiple output for each input packet, but each packet is processed using the same logic.

2. Another pipeline is called the orchestration pipeline. It combines these three stages and processes the packet in multiple parallel paths, performing a different function on copies of the same packet.

The researchers could achieve a degree of modularity by combining two types of pipelines, as shown in figure 2.4.

Another approach to making P4 more extendable and user-friendly was proposed as a language called O4 [34] that, just like previously mentioned work [33], compiles to generic P4. Researchers in this work implement concepts such as loops for creating repetitive P4 code and reducing the size of P4 programs. Researchers in this work did not claim to create a modular P4, but we started our path toward modularity by experimenting with this language. We will discuss our experiments in chapter 4.

CHAPTER 3 Enhancing The Existing Software Switch, Enabling Innovative Solutions

In our research, we initially employed a software switch known as the "simple_switch" behavioral model [12] to simulate and verify our proposed P4 pipeline. In the next chapter, we explore the design of our eCPRI conversion unit, and it is mentioned that this unit uses an enhanced version of BMv2 simple_switch. The present chapter will explore our changes to the existing simple_switch software.

These changes are mostly made to increase the precision of timestamps and enable automatic egress time-stamping, which is necessary for a 3-message PTP-1588 exchange. Moreover, some changes were made to improve the ability of other users to enhance the software switch. These edits mainly involved changing the old style of the codes to newer libraries and languages. Different parts of the software switch using C programming were converted to C++.

Simple_switch has become a widely used architecture for many users due to its similarity to the abstract switch model and widespread use in Mininet [44] environments for testing and development. However, there might be differences in behavior between simple_switch and final hardware implementations, particularly regarding timing and performance. While simple_switch supports many P4 features, some advanced features or hardware-specific optimizations may not be available.

Despite these limitations, simple_switch remains a crucial tool in the P4 developer's toolkit. It provides a means to develop and test P4 programs in a software environment, allowing for rapid iteration and debugging before moving to hardware implementations.

3.1 The architecture of the BMv2 simple switch

BMv2 [12] uses object-oriented C++ programming to implement multiple software switches. Figure 3.1 depicts parent classes for three switches in the BMv2 environment. We focus on the "simple_switch", the most commonly used P4 software switch.

3.2 BMv2 simple switch limitations for implementation of PTP-1588

One of the challenges we faced was the inability to accurately simulate time differences between switches. This limitation arose because the default timestamping method used within

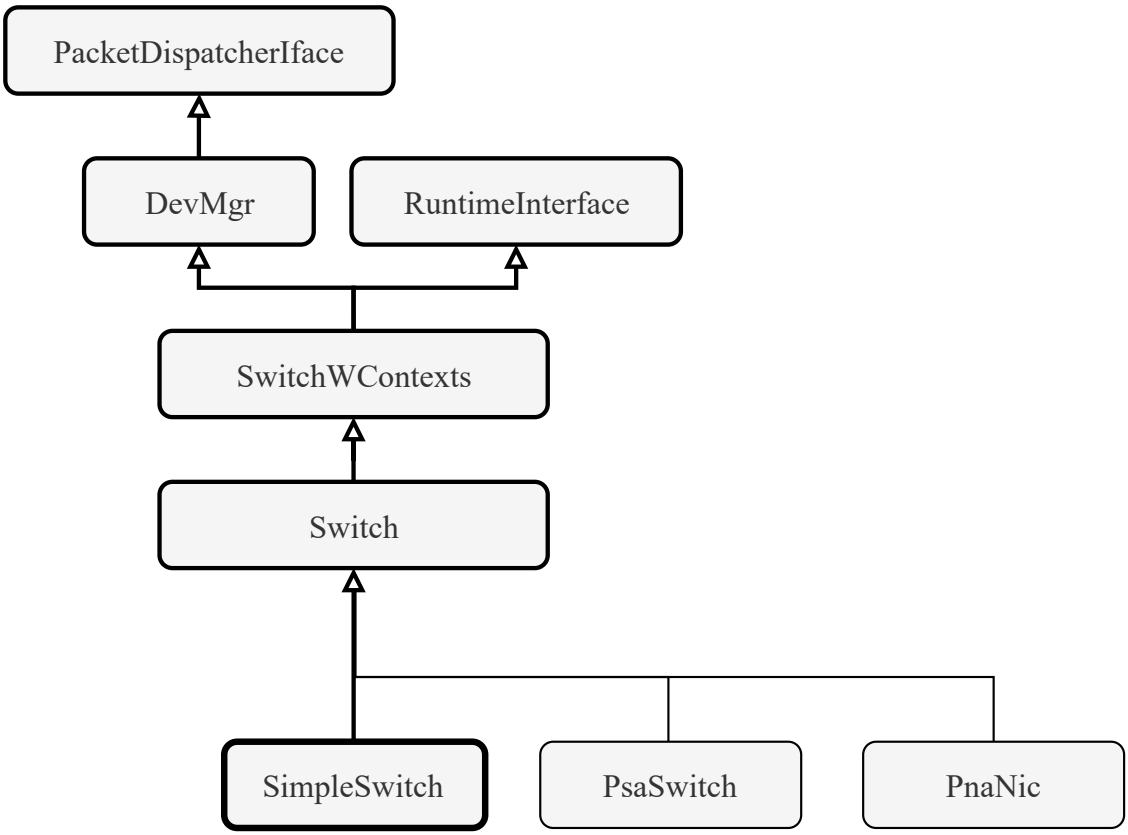


Figure 3.1 Simple switch and other software switches within BMv2 environment inherited from same parent classes.

the simple_switch model was based on the elapsed time since the switch's initialization rather than standard Unix time. As a solution, we introduced new Unix-based timestamps into the model, which enabled us to track real-time events more accurately. Furthermore, we implemented a mechanism to intentionally induce timestamp errors in individual switches, allowing us to simulate and analyze time discrepancies between multiple switches. This enhancement is particularly valuable for testing and validating the behavior of various synchronization protocols and other time-sensitive network protocols, ensuring more accurate and realistic simulations in diverse network scenarios.

3.3 Changes applied to the BMv2 simple switch

Most of the changes we applied to the software switch were intended to increase the precision of ingress and egress timestamp and adopt their format for use in our proposed PTP-1588. We

achieved this goal with complete backward, meaning that none of the existing functionalities or timestamps were removed or changed. Due to the object-oriented design and the lack of provision for future extensions, we had to change many of the parent classes while preserving the correct functionality of the other switches in the BMv2 environment.

3.3.1 New timestamps with better accuracy

In the previous section, we discussed the need for enhanced timestamps. Figure 3.2 shows sections of the code that must be changed to accommodate these new timestamps.

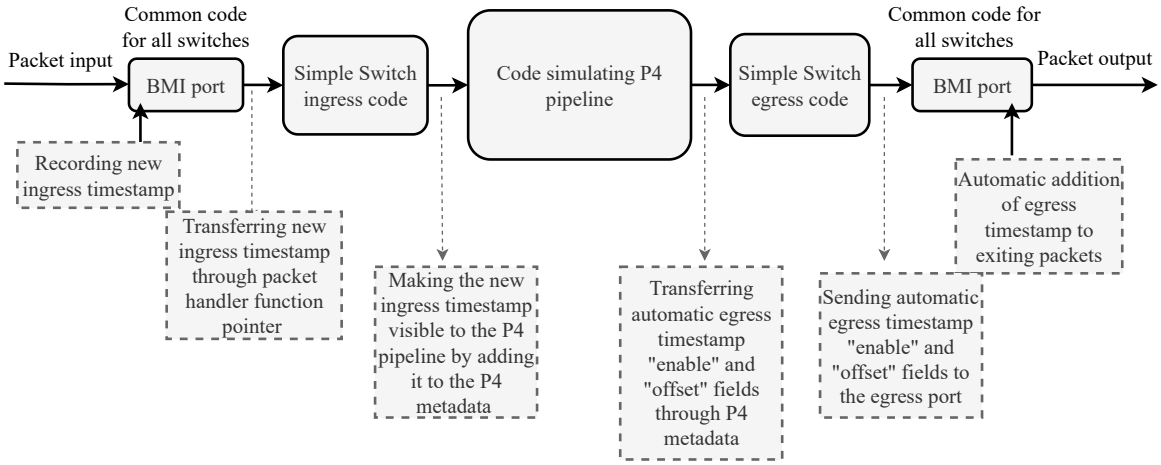


Figure 3.2 Process of transferring timestamps between the P4 pipeline and other sections of the simple_switch code.

To control the behavior of the automatic egress, two metadata fields were created:

1. automatic_truncated_egress_timestamp_enable: A 1-bit field to enable or disable the automatic truncated egress timestamp addition.

2. automatic_truncated_egress_timestamp_offset: A 32-bit field to set the offset in the packet where the egress timestamp will be injected.

A new "packet handler" function was created to allow the transfer of timestamps from "bmi_port" to the next sections of the program.

We verified the functionality of new timestamps using test benches described in chapter 4 where we present the accuracy measurement results for these timestamps.

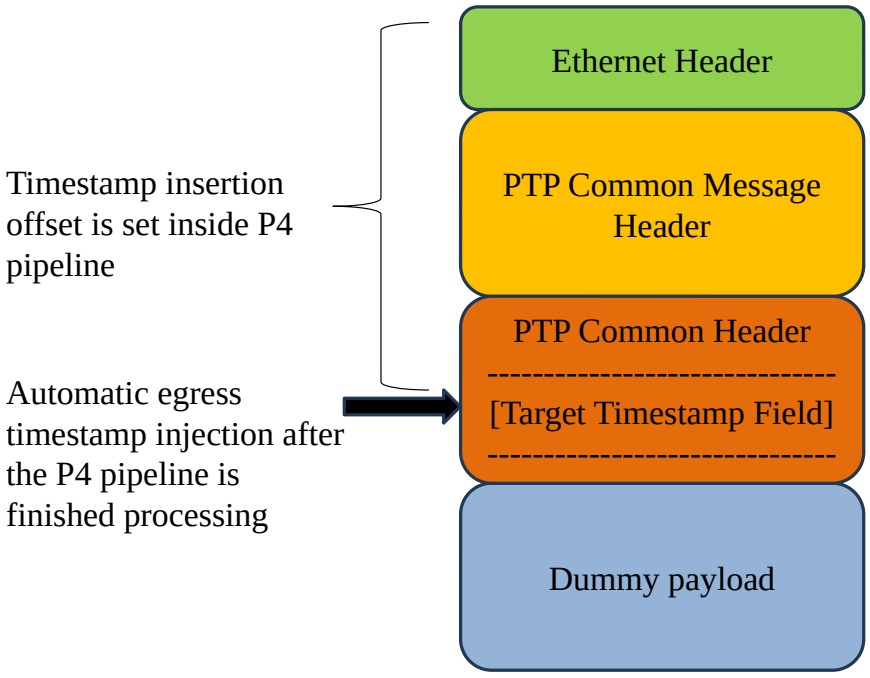


Figure 3.3 Automatic insertion of egress time stamp at the egress thread.

3.3.2 New packet handlers

"packet handler" is a function handler that facilitates the transfer of the packet payload and corresponding metadata through the simple switch. Creating a new "packet handler" function required changing many parent classes. The original packet handler is hardwired to receive only certain variables:

```
using PacketHandler = std::function<void(   int port_num,
                                            const char *buffer,
                                            int len,
                                            void* cookie)>;
```

The new packet handler accepts a struct as an input, making the maintenance and further improvements easier:

```
struct MyMetadata {
    long long ingress_timestamp;
    inline static char shouldBeUsedAsCookie = 0;
    static MyMetadata& castCookie(void* cookie) { return *static_cast
```

```
        <MyMetadata*>(cookie); }
};

struct PacketInfo {
    int port_num;
    const char* buffer;
    int len;
    MyMetadata metadata;
};

using PacketHandlerWithPacketInfo = std::function<void(const
    PacketInfo *packetInfo)>;
```

3.3.3 New set_packet_handler function

The "set_packet_handler" function is responsible for initializing the "device manager" to use the "packet_handler" corresponding to the correct software switch. The old "set_packet_handler" function passes a cookie void pointer to the "SwitchWContexts" object to access and call its specific "receive" function in "bmi_port.c":

```
ReturnCode set_packet_handler(const PacketHandler &handler, void *cookie);
```

The new "set_packet_handler" function uses a lambda to give new "bmi_port.cpp" access to the specific "receive with metadata" function that is defined in the "simple_switch":

```
ReturnCode set_packet_handler_with_packet_info(const
PacketHandlerWithPacketInfo &handler);
```

3.3.4 Creating new tests

To ensure backward compatibility, we ensured that after adding all the new functions and fixing all the errors, the software switch would pass all existing tests. After meeting this goal, we created tests examining new parts of the code.

The original test for the switch did not use the "bmi_port' code. In tests, the "bmi_port" was bypassed using a nanomsg-based injector. Since the "bmi_port" was changed, we created a

libpcap mock that enabled packet injection using the actual BMI code. This approach allowed us to conduct thorough testing and verification across all nine tests in the "simple_switch" tests suite. We achieved successful test outcomes using this method, replacing the existing nanomsg-based injector that typically bypasses the BMI code.

Through this testing process, we discovered two significant bugs in the "bmi_port" module and one in "dev_mgr_bmi.cpp".

3.3.5 Fixing bugs

Bug in bmi_port_destroy_mgr function

The `bmi_port_destroy_mgr` function was designed to destroy all active ports managed by the BMI. However, we discovered that it only removed half of the ports. This issue occurred because elements were removed from the array while iterating through the indices. This operation caused the array size to change dynamically during the loop execution, leading to skipped elements and ultimately leaving specific ports not destroyed. The resolution involved adjusting the iteration mechanism to handle array modifications correctly, ensuring all ports were accurately destroyed as intended.

Deadlock Issue in "_bmi_port_interface_remove" Function

The `_bmi_port_interface_remove` function demonstrated a critical deadlock issue. The problem arose when this function performed a write operation on a pipe and waited for a response via a read while holding the lock `port_mgr->lock`. Concurrently, the `run_select` thread needed to acquire this lock during each iteration of its loop to read from or write to the pipe. This conflict resulted in a deadlock, where neither process could proceed. The solution required modifying the locking mechanism to prevent simultaneous access, allowing both threads to operate without interfering.

Bug in dev_mgr_bmi.cpp Regarding p_monitor

The dev_mgr_bmi.cpp code exhibited a bug similar to a previously encountered issue involving the `p_monitor` thread. Specifically, the `p_monitor` was not being stopped at the correct time, which led to two potential problems:

If `p_monitor` was not halted in the `B̃miDevMgrImp()` destructor, there was a risk of a pure virtual function call occurring. This situation could happen when the thread attempted to execute a method of a derived class after the derived class was destroyed but before the

base class destruction process was completed. Additionally, the `p_monitor` needed to be stopped before destroying `port_mgr`. Failing to do so created a scenario where the thread might attempt to access an already destroyed `port_mgr`, leading to undefined behavior and potential crashes. The solution involved modifying the order of destruction and ensuring that `p_monitor` was properly stopped within the destructor of the `BmiDevMgrImp` class and before the `port_mgr` destruction.

Identifying and resolving these bugs was crucial for ensuring the stability and reliability of the BMI code. These improvements enhanced the robustness of the packet injection and handling processes and contributed to a more reliable P4 runtime environment, ensuring accurate testing and simulation of network packet flows.

CHAPTER 4 ARTICLE 1: 5G FRONTHAUL IN MODULAR P4: ECPRI PROTOCOL PROCESSING AND PRECISE BMV2 TIMESTAMPS FOR PTP-1588

This chapter contains an article with the same title submitted to IEEE Access.

I was the primary contributor to this research, leading to the design, implementation, and testing of the eCPRI packet processing pipeline. I conducted in-depth literature reviews, analyzed experimental results, and drafted most of the paper. I also implemented the majority of the modifications to the BMv2 software switch.

Note to the reader: Readers familiar with chapter 2 may skip section 4.3 of this chapter, as it provides a brief overview of the related work discussed in detail in chapter 2.

Authors: ATABAK NOJAVAN[1], BILL PONTIKAKIS[1], FRANÇOIS-RAYMOND BOYER[2], AND YVON SAVARIA[1], (Life Fellow, IEEE)

Submission date: November 26, 2024

[1]Department of Electrical Engineering, Polytechnique Montréal, Montreal, QC H3T 1J4, Canada; (e-mail: atabak.nojavan@polymtl.ca, bill.pont@gmail.com, and yvon.savaria@polymtl.ca)

[2]Department of Computer and Software Engineering, Polytechnique Montréal, Montreal, QC H3T 1J4, Canada; (e-mail:francois-r.boyer@polymtl.ca)

Corresponding author: Atabak Nojavan (e-mail:atabak.nojavan@polymtl.ca).

4.1 Abstract

P4, a domain-specific language (DSL) for programming network devices, offers flexibility in defining packet processing behaviors. This paper demonstrates the use of P4 to achieve modular eCPRI protocol processing and enhanced PTP-1588 synchronization, both critical for 5G fronthaul applications in Open Radio Access Network (O-RAN) environments. By implementing an eCPRI packet processing unit based on the eCPRI Specification Version 2 and inspired by Intel's FPGA-based IP, we enable the modular addition of new message types and custom packet processing functionality in P4. Our approach reduced lines of code per type by 85% and decreased configuration time by up to 5x compared to traditional methods, significantly simplifying complexity. Additionally, we introduce precise ingress and automatic egress timestamps for the BMv2 software switch to improve PTP-1588 accuracy, reducing er-

ror margins from 24,000 microseconds to 60 microseconds (99.75% improvement) and achieving sub-microsecond precision. Extensive testing in a Mininet environment validates these improvements, demonstrating enhanced precision and flexibility in handling time-sensitive protocols. While this paper focuses on 5G fronthaul applications in O-RAN networks, the techniques and results presented are equally applicable to other use cases across end-to-end 5G networks and beyond, paving the way for modular, high-precision, and programmable solutions in future open and interoperable network architectures.

4.2 Introduction

The primary goal of the Programming Protocol-Independent Packet Processors (P4) language [1,2] is to enable the programmable declaration of forwarding behavior within network devices. Beyond this, P4 empowers network devices to handle complex packet processing tasks traditionally confined to the control plane, creating a Programmable Data Plane (PDP). By unifying control and data plane functions, P4 simplifies the design process and drives innovation in network programmability.

This flexibility stems from P4's support for fundamental data types, like headers, and core functions, such as `extract` and `emit`, which handle packet parsing and deparsing. Packet manipulation is primarily enabled by tables and actions—key constructs that allow packets to be matched, modified, and routed dynamically, with metadata providing supporting information during processing. P4 also supports mathematical operations and externs, enabling the integration of more advanced functionalities beyond its core pipeline.

By implementing packet-processing functionality based on the enhanced Common Public Radio Interface (eCPRI) specification [3] using the P4 language, we aim to achieve greater flexibility, efficiency, and control over its packet-handling behavior. P4 allows us to customize this functionality to specific requirements and optimize data plane performance. Additionally, our approach introduces modular design capabilities in P4, enabling integration with other network components, fostering interoperability, and expanding potential applications. Enhancing P4's flexibility and modularity can increase the functionality's capabilities, reduce dependence on proprietary hardware, and improve its overall adaptability and value. This paper delves into the intricacies of our P4-based implementation of the eCPRI protocol, with applications in the network fronthaul of the Open Radio Access Network (O-RAN) for 5G [8].

In modern cellular networks, Fronthaul refers to the communication network between baseband units (BBUs) and remote radio heads (RHs). With the rise of 5G, network requirements have increased, pushing fronthaul technologies to evolve toward supporting higher

bandwidths, improved efficiency, and greater flexibility. The eCPRI protocol, a standardized protocol in O-RAN, enables these advances by allowing service providers to connect multiple RHs to a centralized BBU, as shown in figure 4.1. This modular architecture fosters vendor diversity and promotes competition, which are key goals of the O-RAN initiative. By utilizing eCPRI, network providers can benefit from a more modular, scalable, and interoperable architecture.

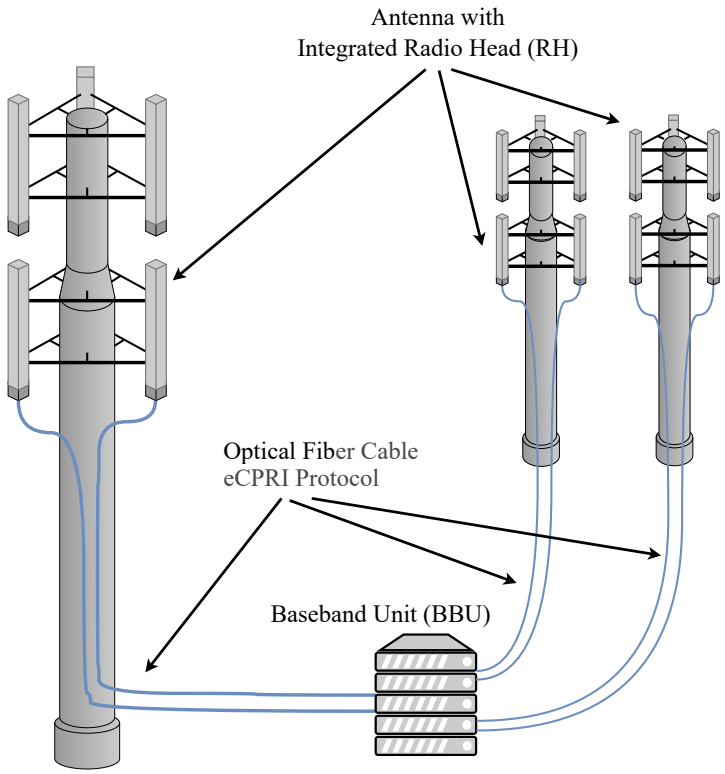


Figure 4.1 Multiple Radio Head (RH) units connected to a centralized BaseBand unit (BBU) using eCPRI links in an Open Radio Access Network (O-RAN).

Guided by Intel's FPGA architectural blocks [4], as we did not have access to the source code of this intellectual property (IP) core, we implemented eCPRI packet processing functionality in P4 based on the eCPRI specification [3], demonstrating how P4 can unify complex packet manipulation and standard forwarding tasks within a single framework. Traditional designs often separate these functions, increasing complexity and potential inefficiencies. Our P4-based approach integrates these functionalities and leverages modular design principles, creating a streamlined solution adaptable to diverse networking needs.

This paper begins by examining an Intel IP core that inspired our design [4], followed by a detailed exploration of our P4 implementation, which includes distinct sections with specific

functionalities. While creating a conversion unit inspired by Intel's IP, we encountered several challenges in adapting certain functionalities to the P4 language, requiring the development of innovative P4 code to address these issues. The four most challenging modules to implement were the:

- *Concatenation (Aggregation) unit*
- *De-concatenation (Deaggregation) unit*
- *Packet sidebands management unit*
- *Synchronization unit*

In addition, a macro preprocessor was repurposed to create a modular design interface that facilitates adding support for new eCPRI message types. To future-proof the protocol, eCPRI provides empty, vendor-specific message types that allow vendors to define additional functionalities as needed. Consequently, any eCPRI implementation should be flexible enough to support these future message types without being hardwired to the existing ones. Our design uses a preprocessor to define header types and necessary functions for each message type within the P4 code to achieve this flexibility. This approach enhances both the flexibility and maintainability of our implementation.

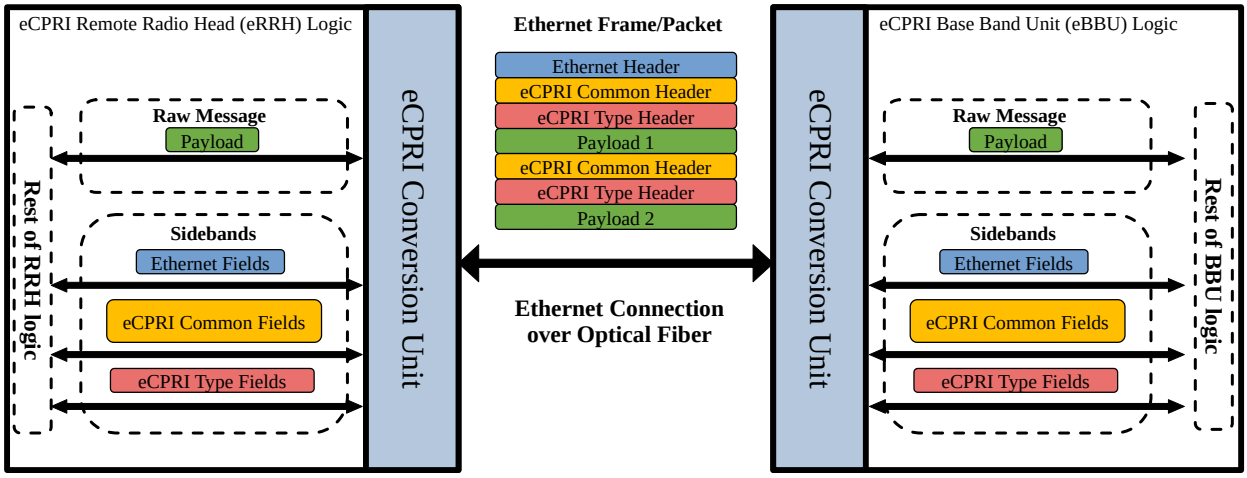


Figure 4.2 eCPRI conversion unit within RRH-to-BBU connection.

Furthermore, the synchronization unit in our implementation detects the clock time difference between two switches using the IEEE 1588 Precision Time Protocol (PTP) standard [35]. To verify and test this unit, we modified existing P4 software switches, such as BMv2, to

incorporate two additional timestamp functions with significantly higher precision than the standard timestamps. These modifications and additional testing capabilities streamline the verification process for our unit and enhance support for time-sensitive network protocols.

4.3 Related work

Robust packet data extraction and storage mechanisms are required to efficiently concatenate and process user data packets, especially for handling variable-length payloads. Previous research has explored various approaches to address these challenges in P4, particularly for applications like deep packet inspection, IoT packet aggregation, and time synchronization. This section reviews existing work on extracting and storing variable-length payloads, implementing time synchronization with PTP-1588 in P4, and non-P4-based implementations of eCPRI packet processing units.

4.3.1 Extracting variable-length packet payload

The current P4 specification only supports parsing and deparsing of variable-length packet headers, and any further processing requires constant-length headers to parse the variable-length packet payload. Several studies have examined this challenge, particularly for applications in Deep Packet Inspection [27–31]. These studies explored implementing variable-length packet payload extraction using fixed-length headers and recirculation. This approach involves repetitive recirculation and parsing with a fixed-length header to extract fields within the packet that vary in size, such as URLs. The algorithm repeats this process until the entire target field (e.g., a URL) is extracted. However, this method introduces latency, as each recirculation cycle adds processing time proportional to the field's length, potentially impacting overall packet processing performance.

4.3.2 Storing variable-length packet payload

Few studies [41, 42] have addressed the challenge of storing variable-length packet payloads by supporting only a limited range of packet sizes, particularly payloads that are multiples of 4 bytes and smaller than 36 bytes, with specific applications in Internet of Things (IoT) packet aggregation. Due to P4's limitations, researchers have explored using multiple one-dimensional registers to create a cluster resembling a two-dimensional array for storing packet payloads. However, P4's lack of native support for two-dimensional registers restricts iteration to a single dimension at a time, requiring compile-time-defined values to access elements in the other dimension. This limitation restricts both the maximum size and expandability

of memory structures.

In our P4-based eCPRI conversion unit, which may be used in BBUs connected to multiple RHs, packets must be aggregated into separate frames based on their destination. This requirement makes the approach in [41, 42] unsuitable for our case. To create a more flexible and scalable implementation, we implemented external memory using an external object to store the extracted packet payloads.

4.3.3 Time synchronization unit

Our time synchronization unit utilizes the IEEE 1588 Precision Time Protocol (PTP), a widely adopted standard for precise clock synchronization. While implementing IEEE 1588 PTP in the data plane using P4 is relatively straightforward and has been explored in previous studies [32], achieving accuracy depends critically on high-precision timestamps. The standard timestamps in BMv2 were unsuitable for IEEE 1588 PTP due to their format and precision limitations. To address these issues, we extended BMv2 by adding new, high-precision timestamps, including an automatic egress timestamp and an option to simulate clock time errors. These enhancements enabled us to implement a 3-message PTP in BMv2 for the first time, allowing for accurate testing of time-sensitive protocols. Measuring egress timestamps at the latest moment in the processing will increase the precision of the PTP-1588 synchronization. Implementing a 3-message PTP will enable the addition of an egress timestamp into the PTP-1588 packets after the pipeline processing is over, increasing the precision.

4.3.4 Non-P4 implementations of eCPRI packet conversion units

In addition to Intel's proprietary FPGA IP core [4], various alternative hardware implementations of eCPRI packet processing units have been developed without relying on P4.

For example, one study [23] implemented an eCPRI packet processing module using a traditional hardware-based approach. This module was deployed on a Zynq FPGA board and integrated with a PetaLinux-based Linux distribution. PetaLinux [37] is a software development kit for embedded Linux systems on AMD (formerly Xilinx) devices, providing an environment for building and deploying Linux images on AMD system-on-chips and FPGAs.

Another study [24] used an Ethernet FPGA IP Core, designed for implementing low-latency Ethernet-based networks for 5G infrastructure, to create an eCPRI over Ethernet conversion unit. In contrast to this proprietary approach, P4 offers an open-source path to network programmability, enabling greater flexibility and faster adaptation to evolving network fronthaul

and eCPRI specifications.

In another hardware implementation of eCPRI over Ethernet [25, 26], researchers developed an Application-Specific Integrated Circuit (ASIC) supporting eCPRI for 5G/LTE hybrid networks. Although this design is highly efficient, the rapidly evolving nature of next-generation fronthaul networks will require a more flexible solution. The P4 language provides a customizable alternative for such applications.

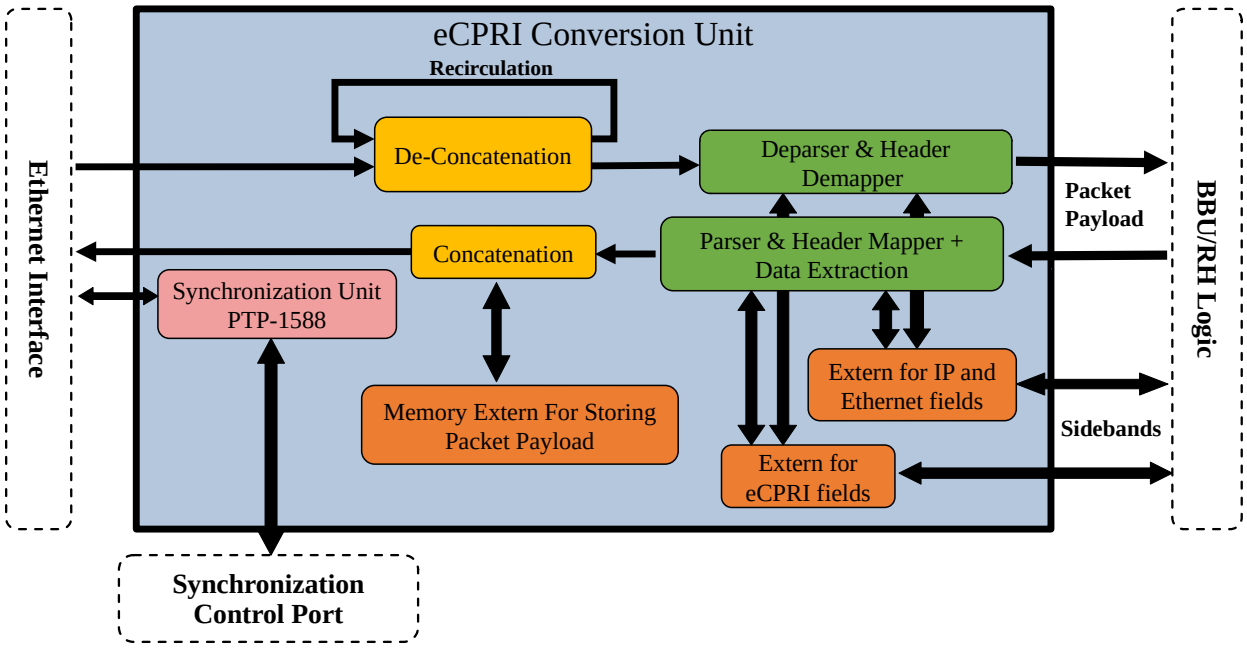


Figure 4.3 Proposed P4-based eCPRI conversion unit and its main interfaces.

4.4 Proposed eCPRI packet conversion unit

Our eCPRI packet conversion unit, developed using the P4 language, draws inspiration from an Intel FPGA IP core [4] and adheres to the eCPRI specification version 2.0. Designed as a fronthaul interface for radio base stations, the packet processing pipeline connects eCPRI radio equipment control (eREC) and radio equipment (eRE) components. It is intended for pairing with custom client logic within FPGA-based smart NICs, as shown in Figure 4.2. The external logic generates raw messages, which the pipeline processes to create eCPRI packets over Ethernet.

In this setup, the baseband unit (BBU) or remote radio head (RH) logic transmits the main message through the primary input channel, while the eCPRI message type and header field values are communicated via sideband signals. The proposed pipeline also populates

Ethernet header fields with values received from these sidebands. It incorporates multiple subsystems for packet processing, including classification, header manipulation, and advanced concatenation/de-concatenation capabilities.

By converting Ethernet frames containing eCPRI messages into parallel data streams, the pipeline facilitates the creation of eCPRI interfaces for FPGA-based 5G fronthaul networks, connecting eREC to eRE over a fronthaul transport network. For instance, when eCPRI message type 0 enters the pipeline from the client logic side, the IQ_data (the main payload) and other header fields, such as pc_id and seq_id, are transmitted through dedicated input streams. The pipeline then populates the corresponding header fields based on these sideband inputs

While implementing our P4-based eCPRI conversion unit, we encountered challenges stemming from inherent limitations in P4's design as a packet-forwarding language, which focuses on forwarding behavior rather than direct manipulation of packet payloads. To overcome these constraints, we developed innovative solutions to address several key issues, including:

1. Lack of support for sidebands: P4 pipelines are designed to operate on discrete packets, limiting their ability to process sideband signals or parallel data streams. This limitation arises from P4's role as a standalone packet-processing pipeline, which complicates integration with hardware components relying on parallel data transmission. The requirement for packet-based communication can introduce additional complexity and potential performance bottlenecks in such scenarios. To address these challenges, we developed an external sideband management system, incorporated into the parser and header mapper unit through the use of P4 externs. Figure 4.3 illustrates the P4 pipeline implementation, with the sideband section highlighted. A detailed discussion of this unit is provided in Section 4.4.1 on the parser and header mapper.

2. Concatenation and de-concatenation: While packet concatenation can reduce network overhead by combining smaller packets into larger frames, P4's primary focus on forwarding complicates handling packet payloads for this purpose. However, using external structs as sidebands provides a solution. By leveraging these external structures, the P4 pipeline can interact with external memory or storage systems to efficiently perform packet concatenation and de-concatenation operations. Similar to our solution for P4's sideband limitation, we mitigated the concatenation issue using P4's externs while exploring two other approaches. Furthermore, we used packet recirculation as a workaround to implement de-concatenation. Our specific solutions are discussed in sections 4.4.3 and 4.4.4.

4.4.1 Parser and header mapper

The primary function of a P4 parser is to extract and parse headers from incoming packets. This part of the pipeline defines a finite state machine that traverses an incoming byte stream, identifying and extracting headers based on a predefined sequence. The parser operates on a packet_in object, representing an incoming packet, and determines valid header sequences, extracting each header and its fields as specified.

In P4, the entire packet is received as a byte stream, with no built-in mechanism to distinguish between headers and payload. It is up to the P4 code to identify headers by examining type fields in previously parsed headers. For example, if the input port is an Ethernet port, the parser will first parse the Ethernet header, examine its type field, and use that information to determine the protocol and size of the next header. This process continues step-by-step, parsing each subsequent header based on the extracted type fields.

Our parser is designed for two-way packet conversion between the Ethernet and external logic ports, handling packets from both the eCPRI logic side and the Ethernet interface. The parser extracts the Ethernet and IP headers for packets received on the Ethernet port using basic P4 parsing functions.

However, handling packets from the eCPRI logic side requires more innovative methods. These packets do not contain headers, so the necessary header field values are transmitted via sidebands. We recreated these sidebands using C++ externs to retrieve the header values. When a raw message is received from the eCPRI logic side, the parser invokes C++ functions to obtain the required values, which are then used to populate the Ethernet and eCPRI headers within the packet.

In a potential future hardware implementation, these externs could be realized using a hardware description language, enabling the P4 pipeline to interface with external circuits that generate eCPRI messages and provide sideband data. Our parser unit is designed to include a packet data extraction component to support such functionality.

4.4.2 Packet payload extraction

P4 does not provide a direct mechanism for handling packet data, as it can only parse bits as headers. This limitation posed a substantial challenge since our pipeline must support packet concatenation and de-concatenation for the eCPRI conversion unit. Packet concatenation is a network optimization technique that combines multiple smaller packets into a single larger packet, reducing overhead at the physical and link layers [45, 46].

To implement concatenation, packet data must be extracted from multiple packets and com-

bined into a larger frame, which can save bandwidth and improve efficiency, particularly with smaller packets. However, implementing concatenation and de-concatenation mechanisms requires the ability to extract and store packet data, which P4 lacks. To work around this limitation, packet data are parsed as headers. This is feasible because packet headers and payloads are transmitted as a unified byte stream with no intrinsic distinction between them.

P4 supports both variable and constant-length header types. Although packet data can have arbitrary lengths, making variable-length headers seemingly more suitable, they cannot be manipulated; P4 only allows parsing and deparsing them. Therefore, variable-length headers cannot be used to extract packet data for storing (in registers or extern memory), leaving constant-length headers as the only option.

The data extraction unit employs a dynamic approach to accommodate byte streams of varying lengths within fixed-size storage types to address the limitation of using only fixed-size headers for payload parsing. By first determining the total byte count, the unit strategically selects a combination of storage units whose sizes collectively match the desired length. This parsing method ensures that the entire byte stream is captured without exceeding storage limits, making it adaptable for different data sizes. The header sizes follow a geometric progression, starting at 1 byte and doubling up to 256 bytes. This technique allows for parsing packet data across a wide range of sizes, from 1 to 511 bytes. Figure 4.4 shows an example of a 14-byte payload parsed using headers with 2, 4, and 8-byte lengths.

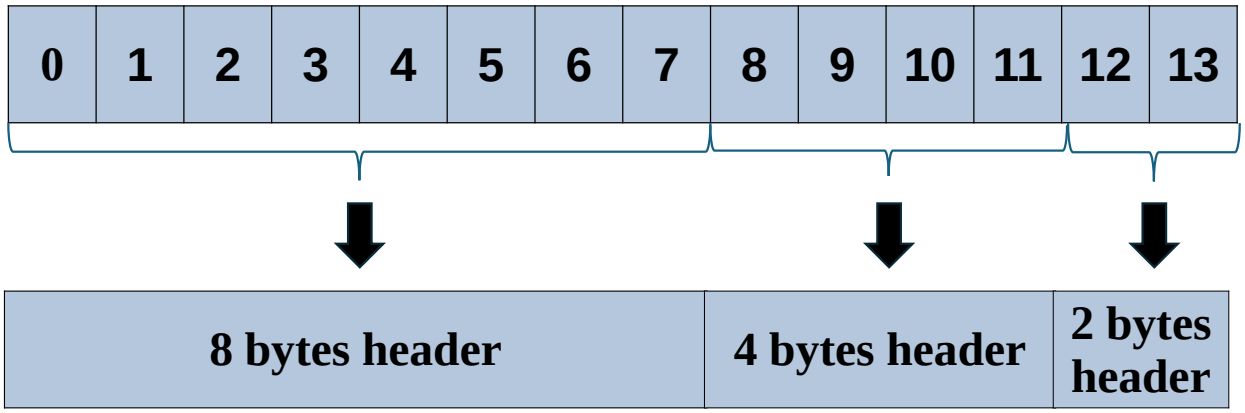


Figure 4.4 Example of dynamic data extraction on a 14-byte payload using 8, 4, and 2-byte headers.

A portion of the P4 code for payload extraction is as follows:

```
bit<32> a = standard_metadata.
```

```
    packet_length;
if (a[0:0] == 1)
    packet.extract(hdr.data_1);
if (a[1:1] == 1)
    packet.extract(hdr.data_2);
if (a[2:2] == 1)
    packet.extract(hdr.data_4);
if (a[3:3] == 1)
    packet.extract(hdr.data_8);
if (a[4:4] == 1)
    packet.extract(hdr.data_16);
if (a[5:5] == 1)
    packet.extract(hdr.data_32);
```

The code above demonstrates a dynamic payload extraction method using headers of varying sizes. Based on the total packet length, it selectively extracts headers to accommodate different byte lengths.

P4-programmable ASICs often have constraints on the size and number of headers they can process, due to fixed hardware resources and predefined parsing capabilities. In contrast, FPGA implementations offer greater flexibility in defining and processing custom headers, as their reconfigurable nature allows for tailored parsing logic and resource allocation. This adaptability makes FPGAs well-suited for applications like the Intel eCPRI IP and the proposed P4-based eCPRI conversion unit, which are designed to create Ethernet interfaces and integrate into larger systems.

4.4.3 Concatenation unit

Packet concatenation enhances efficiency by reducing the number of headers, thereby saving bandwidth and enabling faster transmission times [45, 46]. Figure 4.5 illustrates a simple example of concatenating two Ethernet packets carrying eCPRI payloads, as specified in the eCPRI standard [3].

When concatenating Ethernet packets containing eCPRI messages, the eCPRI conversion unit combines those with identical source and destination Ethernet addresses. Each eCPRI payload includes three essential fields: length, type, and the '`C`' bit. The 'length' field specifies the size of the individual eCPRI packet, the 'type' field indicates the specific header type within the eCPRI payload, and the '`C`' bit signals the presence of additional eCPRI packets

within the Ethernet frame.

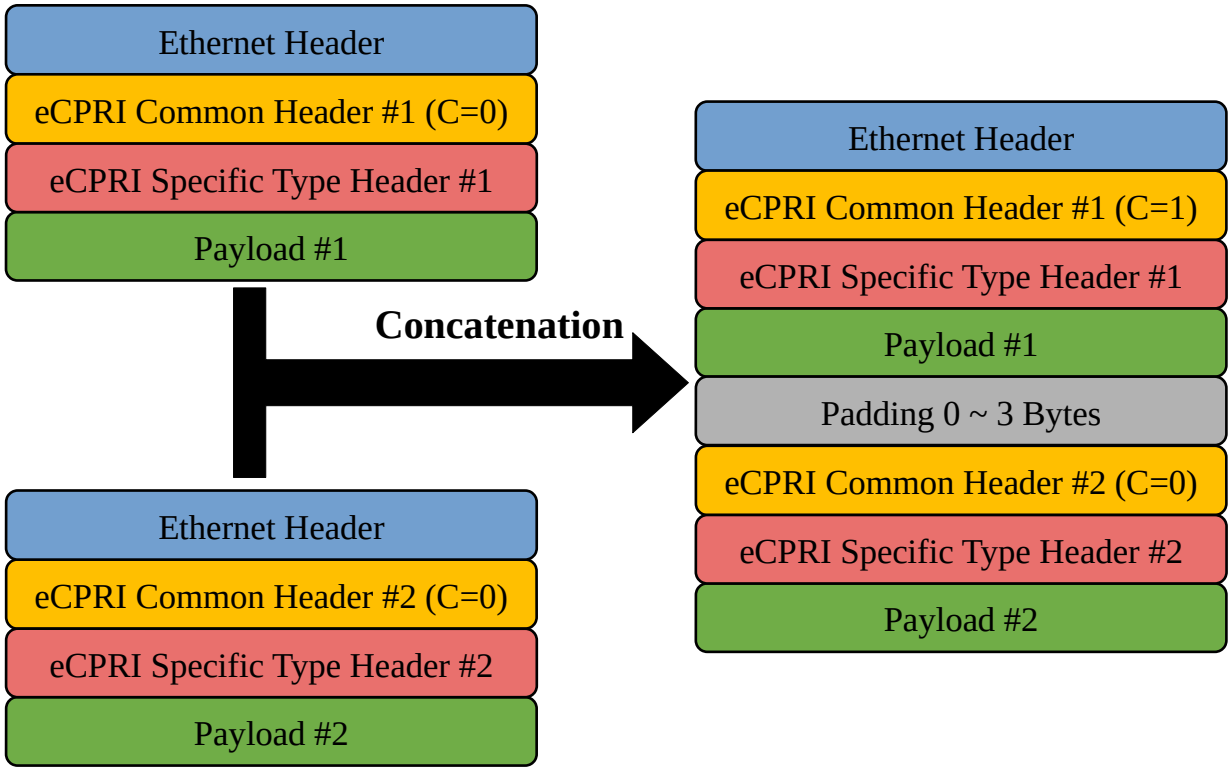


Figure 4.5 Concatenation of two separate eCPRI packets into a single Ethernet frame.

A mechanism for accessing previously received packets is required to perform packet concatenation. However, P4's stateful capabilities are limited to registers, counters, and meters, making direct packet storage infeasible. As a workaround, the pipeline creates frames and continuously recirculates them, adding newly arrived packets to each frame. Once a frame is full, recirculation stops, and the completed frame is transmitted to the output port, eliminating the need for storing large frames.

To implement packet concatenation in P4, we explored three mechanisms:

1. **Recirculation**: Using P4's recirculation mechanism to pass packets through the pipeline multiple times, adding each new packet to the frame.
2. **Standard Registers**: Utilizing standard P4 registers as temporary packet storage, although limited by size and structure.
3. **Externs**: Leveraging externs for packet data storage, which proved most suitable for our needs.

We implemented concatenation using two distinct methods: recirculation and externs, while also exploring the use of P4 registers as payload memory. As discussed in the following sections, externs provided the highest efficiency and lowest latency, making them the most practical option to implement.

Using P4's recirculation mechanism to concatenate packets

To avoid a complicated frame storage process, our first design approach employs continuous recirculation, allowing the P4 pipeline to maintain access to the frame until it is complete. As illustrated in Figure 4.6, the concatenation process follows a logical sequence involving recirculation mechanisms.

Upon receiving a new packet, the system first determines whether an incomplete frame is in recirculation progress. If no such frame exists, the P4 pipeline initiates recirculation of the new packet as the foundation of a new frame. If an incomplete frame in recirculation exists, the P4 pipeline stores it in registers and appends it to the frame upon its arrival. The P4 pipeline checks if the frame is full and recirculates it if more packets are required. This cycle continues until the frame is complete, at which point it is sent to the output port.

For each packet added to the frame, Figure 4.6 provides a detailed flow chart of the decision-making process, including checks for saved packets, frame capacity, and recirculation status. This structured approach ensures efficient frame formation while maintaining pipeline simplicity.

As illustrated in Figure 4.7, the recirculation mechanism progresses across distinct time instances ($t = t_1$ to $t = t_4$). At $t = t_1$, the first packet is received and re-circulated as the starting point of a new frame. When subsequent packets of the desired type arrive, the P4 pipeline processes their data, interprets them as a large header using previously described data extraction techniques, and temporarily stores the payload in registers. At $t = t_2$, $t = t_3$, and $t = t_4$, these packets are appended to the existing frame during recirculation. This process repeats until the frame reaches its predetermined size limit. Once the frame is full, it is sent to the output port, ready for transmission.

Storing a single packet's payload in registers at any given time, as performed in this approach, is relatively straightforward; however, as discussed in the next section, storing entire frames with multiple packets can strain register capacity, limiting their suitability for this purpose.

This implementation avoided complex frame storage mechanisms by utilizing recirculation to keep the frame accessible within the P4 pipeline for further processing. However, it introduced limitations:

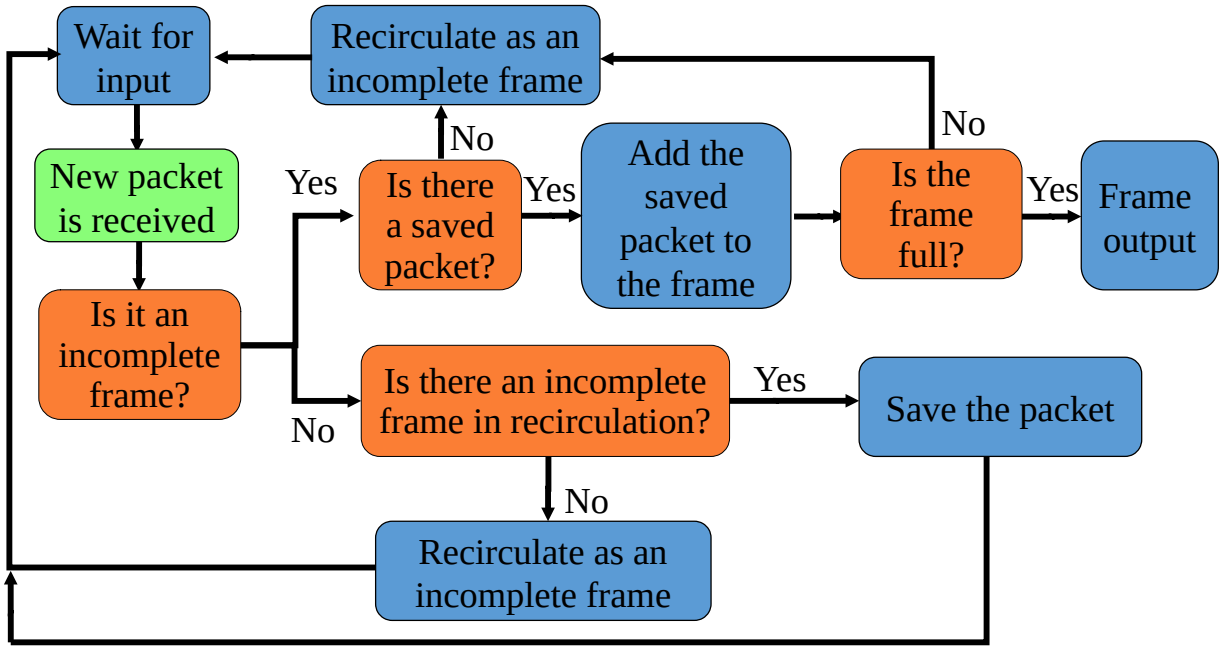


Figure 4.6 Flowchart of the concatenation algorithm using recirculation and cloning.

1. **Increased overhead and low efficiency:** Repeatedly recirculating the frame back to the input port consumes bandwidth and increases traffic in the input port buffer. Additionally, if the next suitable packet for concatenation arrives late, the frame may spend an extended time recirculating, which increases the forwarding delay for packets already in the frame. Meanwhile, if multiple eligible packets arrive while a frame is still recirculating, the P4 pipeline's limited storage capacity forces it to send these packets individually without concatenation, reducing efficiency.

2. **Implementation complexity and lack of extensibility:** The stateful nature of this approach requires maintaining a record of the recirculating frame, adding complexity to the implementation. Creating a concatenation unit capable of handling multiple frames simultaneously would introduce even greater challenges, limiting the scalability of this approach.

3. **Packets out of order:** The sequential addition of packets to the frame in reverse order results in an incorrect packet sequence, leading to potential issues at the receiving end. Out-of-order packet extraction may cause packet drops in protocols that require strict sequencing.

To address these issues and inefficiencies, we explored two alternative solutions, which are

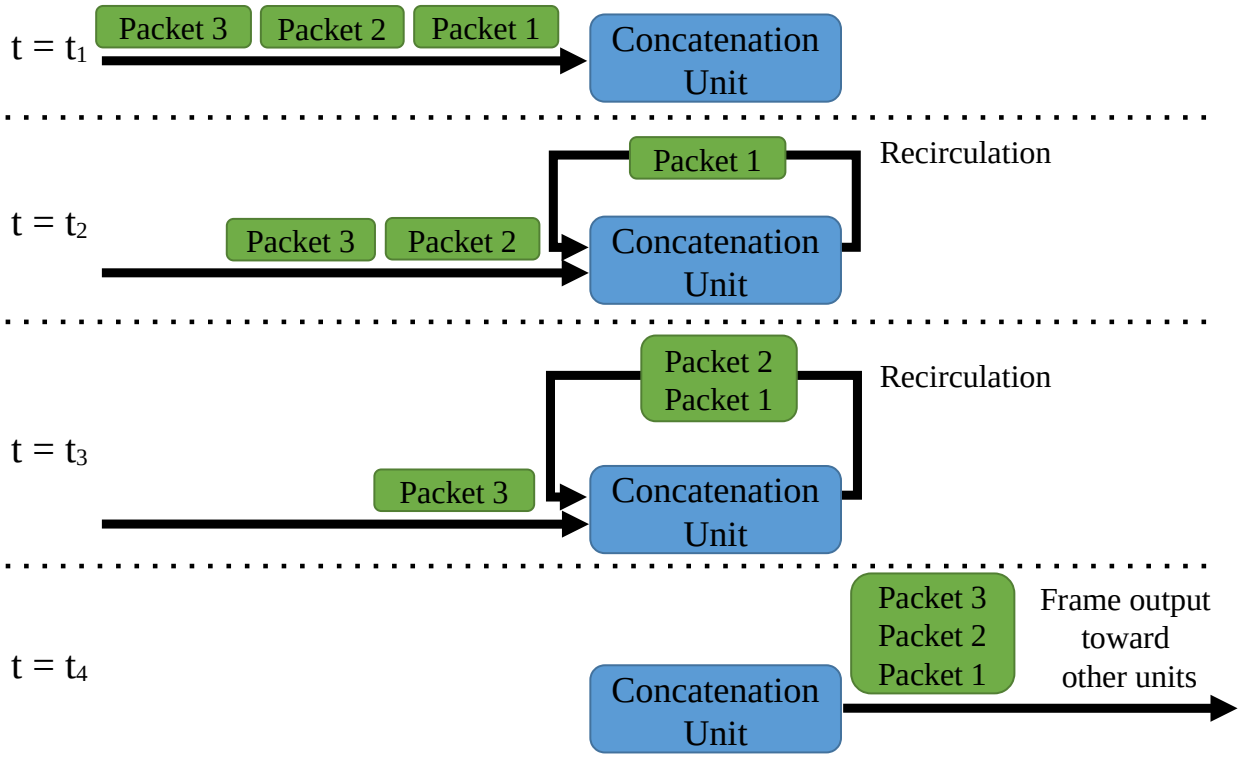


Figure 4.7 Simplified representation of the concatenation process using recirculation, illustrated across distinct stages ($t = t_1$ to $t = t_4$)

discussed in the following subsections.

Using standard P4 registers as packet storage to concatenate packets

Following the approach implemented in another study [42], we considered using P4 registers for packet storage as a potential solution for concatenation. However, this approach has several significant limitations.

While some P4 target hardware restricts the use of excessively large register types, this limitation does not apply to our FPGA-based design. However, P4 register arrays are inherently one-dimensional and have fixed, compile-time-defined sizes. This constraint limits their ability to accommodate variable-length packet payloads, which can vary significantly between packets.

Since each packet payload is extracted using multiple headers (see Figure 4.4), and each frame may contain multiple packets, an efficient storage implementation would need to iterate through two dimensions: one for the packets and one for the headers within each packet.

These limitations become even more pronounced when managing multiple frames simul-

taneously, especially for packet concatenation based on destination addresses. The one-dimensional structure of P4 registers requires defining separate register arrays for each frame, leading to code redundancy, reduced readability, and increased maintenance overhead. This approach ultimately hinders scalability, making it impossible to efficiently handle complex scenarios involving multiple concatenated packets. To overcome these challenges, we propose leveraging external memory for packet data storage. This approach is discussed in detail in the following subsection.

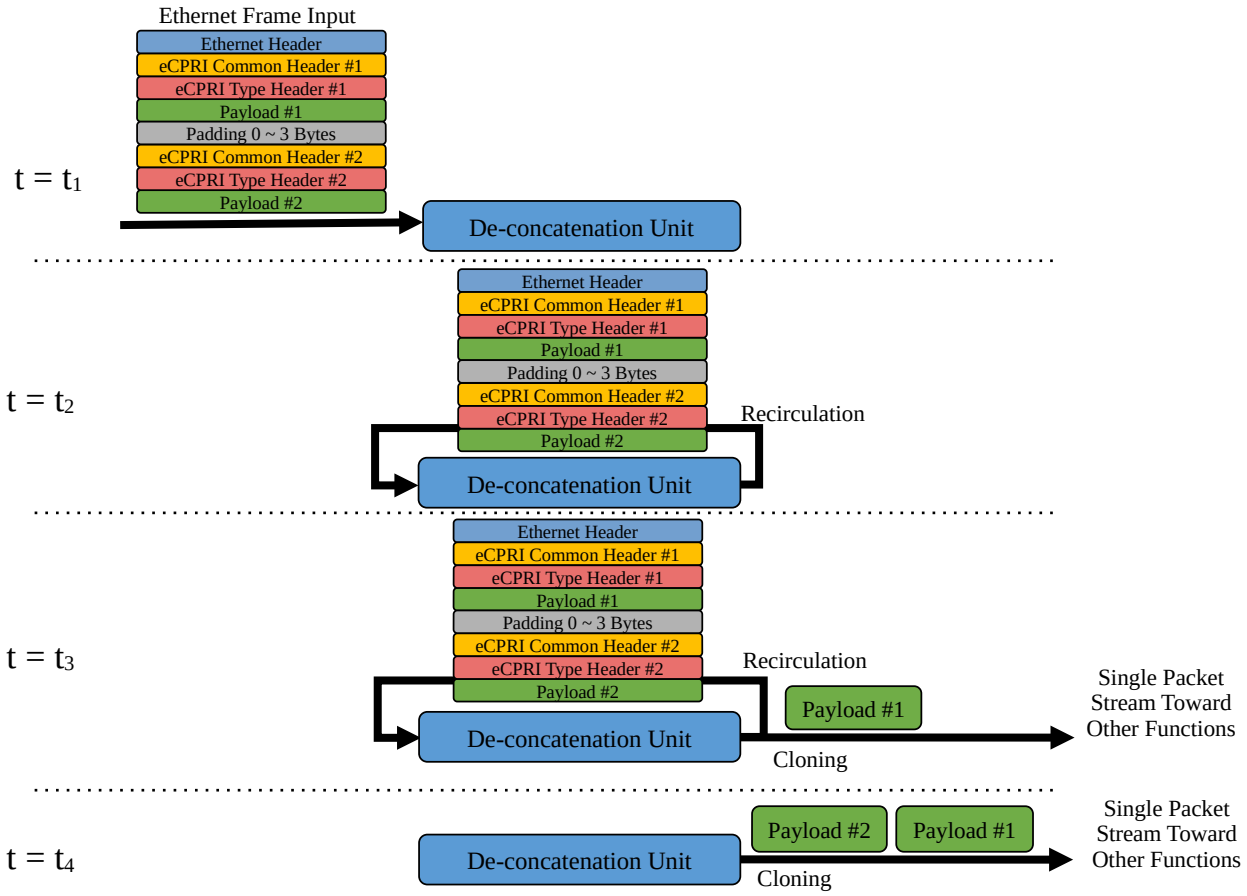


Figure 4.8 Stages of the de-concatenation process on an Ethernet frame illustrate recirculation and cloning. The parallel sidebands for header field transmission are not shown.

Using externs for packet data storage

After extracting packet data into headers, the P4 pipeline can pass these headers to extern functions for storage in memory. In the software environment, this memory is simulated using a file, with the extern function implemented in C++. The extern object `frame1` stores the assembled frame. During this process, the newly received payload is marked for dropping, as

it is saved in a frame and should not be forwarded to the output port.

```
mark_to_drop(standard_metadata);
```

When concatenating multiple eCPRI messages, padding bytes (0 to 3) are inserted before the payload of subsequent messages to ensure proper alignment and prevent data corruption. This alignment guarantees correct processing by the receiving entity. If the current packet is not the first packet in the frame, padding bytes are added before the payload.

```
frame1.add_padding();
```

Next, the correct eCPRI headers are read from sidebands and added to the frame.

```
frame1.add_ecpri_common_header(
    data_size_var,
    meta.sideband.ecpri_type);
```

For instance, if the packet is of type 3, the eCPRI message type 3 header is added.

```
if (meta.sideband.ecpri_type == 3){
    hdr.ecpri.size = hdr.ecpri.size + 8;
    frame1.save(meta.sideband.
        type3_pc_id);
    frame1.save(meta.sideband.
        type3_seq_id);
}
```

Finally, the extracted payload is saved into the frame.

```
if (hdr.data_1.isValid())
    frame1.save (hdr.data_1.data);
if (hdr.data_2.isValid())
    frame1.save (hdr.data_2.data);
if (hdr.data_4.isValid())
    frame1.save (hdr.data_4.data);
if (hdr.data_8.isValid())
    frame1.save (hdr.data_8.data);
if (hdr.data_16.isValid())
```

```
    frame1.save (hdr.data_16.data);
if (hdr.data_32.isValid())
    frame1.save (hdr.data_32.data);
```

When the frame reaches a user-defined size, it is emitted to an output port with an Ethernet header added. By carefully arranging padding bytes, headers, and payloads, the concatenated Ethernet frame adheres to established standards, ensuring successful transmission and processing.

Our work is limited to software implementation and verification. However, a potential hardware implementation could utilize hardware description languages, such as VHDL, or C++ code translated to hardware through high-level synthesis, to realize this memory and its extern function.

4.4.4 De-concatenation unit

To facilitate packet de-concatenation, the proposed unit employs recirculation and cloning mechanisms (as illustrated in Figure 4.8). Recirculation enables processing of each individual packet within the frame, while cloning generates the empty packets required for de-concatenation.

Unlike packet concatenation, de-concatenation through recirculation introduces minimal performance overhead. Since all constituent packets are already present within the frame, they can be sequentially extracted without significant added latency.

Upon receiving an Ethernet frame containing eCPRI messages ($t = t_1$), the parser processes the Ethernet and eCPRI common headers. The P4-based eCPRI conversion unit then recirculates the frame to the cloning port ($t = t_2$). To extract the first eCPRI packet, the unit parses the header and data fields. To avoid unnecessary data storage or external transfers, the pipeline leverages variable-length headers, which P4 supports for extraction and emission. Although P4 does not support additional operations on these headers, our requirement of re-emitting the header into a single packet aligns with its capabilities.

The P4-based eCPRI conversion unit examines the header field of the extracted eCPRI packet to check if additional packets are present within the frame. If the '`C`' bit is set to 1, this indicates the presence of at least one additional eCPRI message. In such cases, the unit recirculates the frame for further processing ($t = t_3$). Meanwhile, the extracted packet is inserted into an empty cloned packet and forwarded to the subsequent stages of the conversion unit. The recirculation process will stop when all packets are extracted ($t = t_4$).

Upon arrival at the cloning port, the conversion unit utilizes the `packet.advance` command to skip previously extracted packets and directly access the next packet within the frame. The implementation processes padding bytes, ranging from 0 to 3, to ensure correct alignment. This alignment requirement mandates that each packet's offset, relative to the beginning of the eCPRI common header, be a multiple of 4 bytes. Figure 4.9 illustrates a frame containing nine different eCPRI packets, while Figure 4.10 shows the resulting stream of single packets containing only eCPRI payloads after de-concatenation.

```
 Packet received at port 1 :
0000  FF FF FF FF FF FF EE EE EE EE EE EE AE FE 11 03  ................
0010  00 16 12 34 56 78 12 34 56 78 41 61 61 61 61 61  ...4Vx.4VxAaaaaa
0020  61 61 61 61 61 61 61 41 00 00 11 04 00 37 12 34  aaaaaaaA.....7.4
0030  56 78 9A BC CB A9 87 65 00 2B 42 62 62 62 62 62  Vx.....e.+Bbbbbb
0040  62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62  bbbbbbbbbbbbbbbb
0050  62 62 62 62 62 62 62 62 62 62 62 62 62 62 62 62  bbbbbbbbbbbbbbbb
0060  62 62 62 62 42 00 11 06 00 1A 12 34 56 43 63 63  bbbbB......4VCcc
0070  63 63 63 63 63 63 63 63 63 63 63 63 63 63 63 63  cccccccccccccccc
0080  63 63 63 43 00 00 11 07 00 0C 12 00 56 01 9A BC  cccC........V...
0090  00 19 87 65 43 21 11 02 00 14 12 34 56 78 45 65  ...eC!.....4VxEe
00a0  65 65 65 65 65 65 65 65 65 65 65 65 65 45 11 01  eeeeeeeeeeeeeE..
00b0  00 12 12 34 56 78 46 66 66 66 66 66 66 66 66 66  ...4VxFfffffffff
00c0  66 66 66 46 00 00 11 01 00 17 12 34 56 78 47 67  fffF.......4VxGg
00d0  67 67 67 67 67 67 67 67 67 67 67 67 67 67 67 67  gggggggggggggggg
00e0  47 00 11 08 00 25 12 34 56 12 12 34 56 78 12 48  G....%.4V..4Vx.H
00f0  68 68 68 68 68 68 68 68 68 68 68 68 68 68 68 68  hhhhhhhhhhhhhhhh
0100  68 68 68 68 68 68 68 68 68 68 48 00 00 00 11 0A  hhhhhhhhhH.....
0110  00 1B 12 34 56 78 49 69 69 69 69 69 69 69 69 69  ...4VxIiiiiiiiii
0120  69 69 69 69 69 69 69 69 69 69 69 69 49 00 10 0B  iiiiiiiiiiiiI...
0130  00 26 12 34 56 78 CC CC CC CC DD DD DD DD 4A 6A  .&.4Vx........Jj
0140  6A 6A 6A 6A 6A 6A 6A 6A 6A 6A 6A 6A 6A 6A 6A 6A  jjjjjjjjjjjjjjjj
0150  6A 6A 6A 6A 6A 6A 6A 4A                          jjjjjjjJ
```

Figure 4.9 A standard Ethernet frame containing multiple eCPRI packets and padding bytes between packets entering the de-concatenation process.

To maintain a stateless de-concatenation process, metadata within the packet is used to track the frame's state. This stateless approach is advantageous given potential recirculation delays and the possibility of receiving multiple frames simultaneously, ensuring that the unit can effectively de-concatenate multiple frames without losing track. By avoiding the use of registers for state storage, we enhance the switch's robustness as the metadata recirculates alongside the packet, enabling parallel de-concatenation.

In contrast to concatenation, recirculation introduces minimal delay in the de-concatenation process. Since de-concatenation is mainly independent of incoming packets, it can be effi-

ciently executed using consecutive recirculation operations when the cloning port is given higher priority.

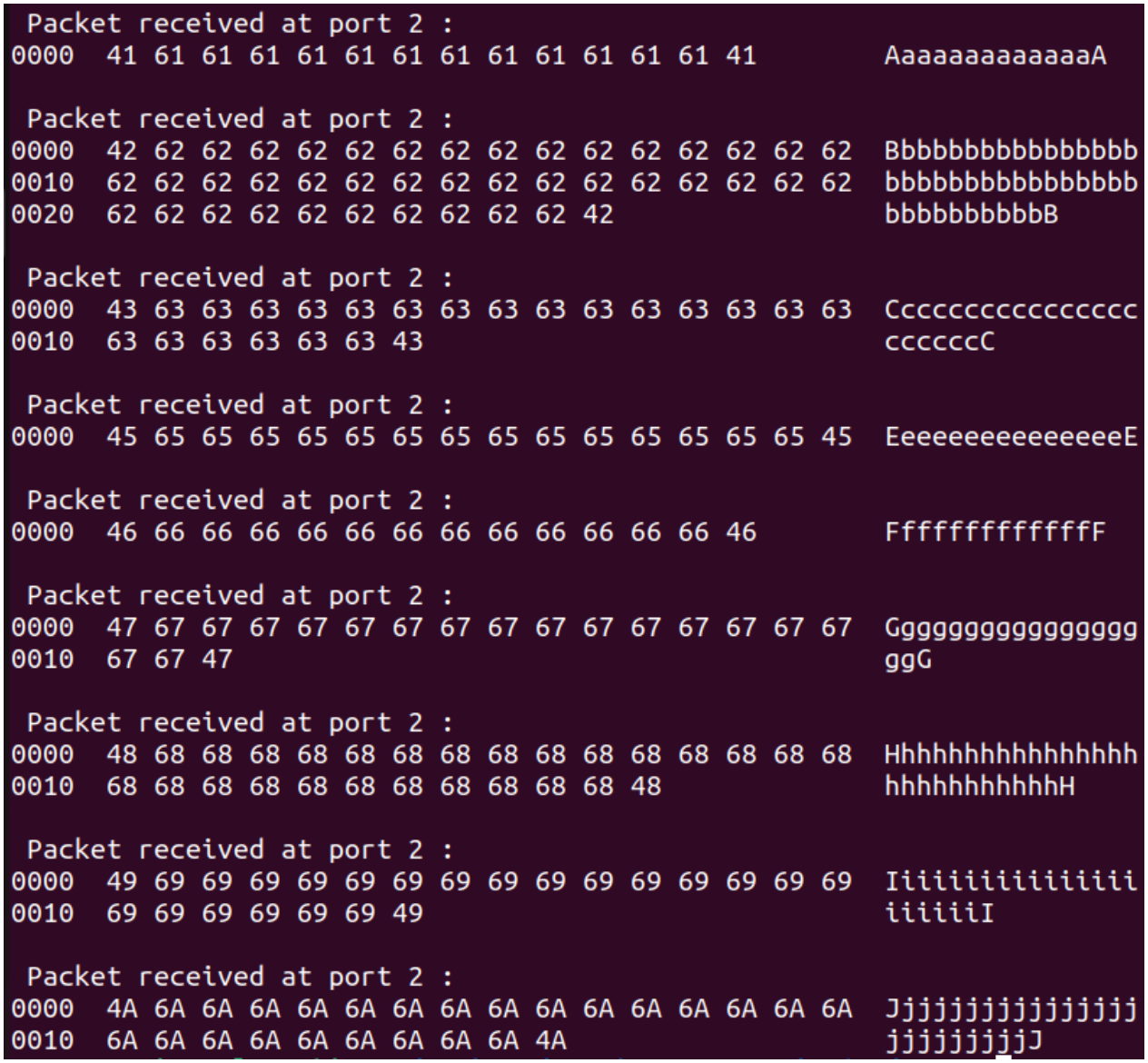


Figure 4.10 Stream of single packets containing only eCPRI data exiting the switch after de-concatenation.

The ability to de-concatenate eCPRI frames is essential for ensuring interoperability with other network devices. While an eCPRI packet processing unit may optionally forgo concatenation, it is crucial to support de-concatenation to handle frames that other devices have already concatenated.

4.4.5 Clock synchronization unit

Precise network clock synchronization is crucial for maintaining consistency in distributed databases, e-commerce applications, and datacenter environments [14]. In advanced cellular networks like 5G, which employ techniques such as Multiple-Input Multiple-Output (MIMO), Carrier Aggregation (CA), and Coordinated Multi-Point (CoMP), accurate time synchronization is even more critical. Base stations and backhaul equipment in these networks require precise timing to coordinate transmissions and prevent interference [15].

Traditional time synchronization protocols, such as PTP, are often implemented in the control plane, which limits their precision for modern, in-network computing applications [32]. By leveraging programmable switching ASICs, P4 allows PTP-1588 to be implemented directly in the data plane, often called Ethernet-based PTP-1588. This approach enhances accuracy and reduces error rates. Additionally, it simplifies the synchronization process, as no external entities other than the switch are involved.

The PTP unit generates new PTP messages, handles incoming PTP messages through the Ethernet port, processes data extracted from PTP-1588 protocol packets, and generates the final compensation values. Depending on configuration and system requirements, PTP-1588 exchanges can involve 3 or 4 messages.

In a 3-message exchange, illustrated in Figure 4.11, the grandmaster clock sends a Sync message with an egress timestamp to the subordinate clock, followed by a Delay Request message from the subordinate clock to the grandmaster clock. The grandmaster responds with a Delay Response message that includes the ingress timestamp of the Delay Request message, thus completing the exchange.

Figure 4.12 shows a 4-message exchange, where an additional Follow-Up message is sent from the grandmaster clock to the subordinate clock. This Follow-Up message carries the egress timestamp of the Sync message and is typically used when an egress timestamp cannot be added directly to the Sync message. With simple calculations, the clock offset between two switches can be determined using the following equations:

$$\begin{aligned} T2 - T1 &= delay_{S1->S2} + \text{clock\_offset}. \\ T4 - T3 &= delay_{S2->S1} - \text{clock\_offset}. \end{aligned} \tag{4.1}$$

Assuming the delay between the two switches is symmetrical, we can derive equation (4.2) from equation (4.1).

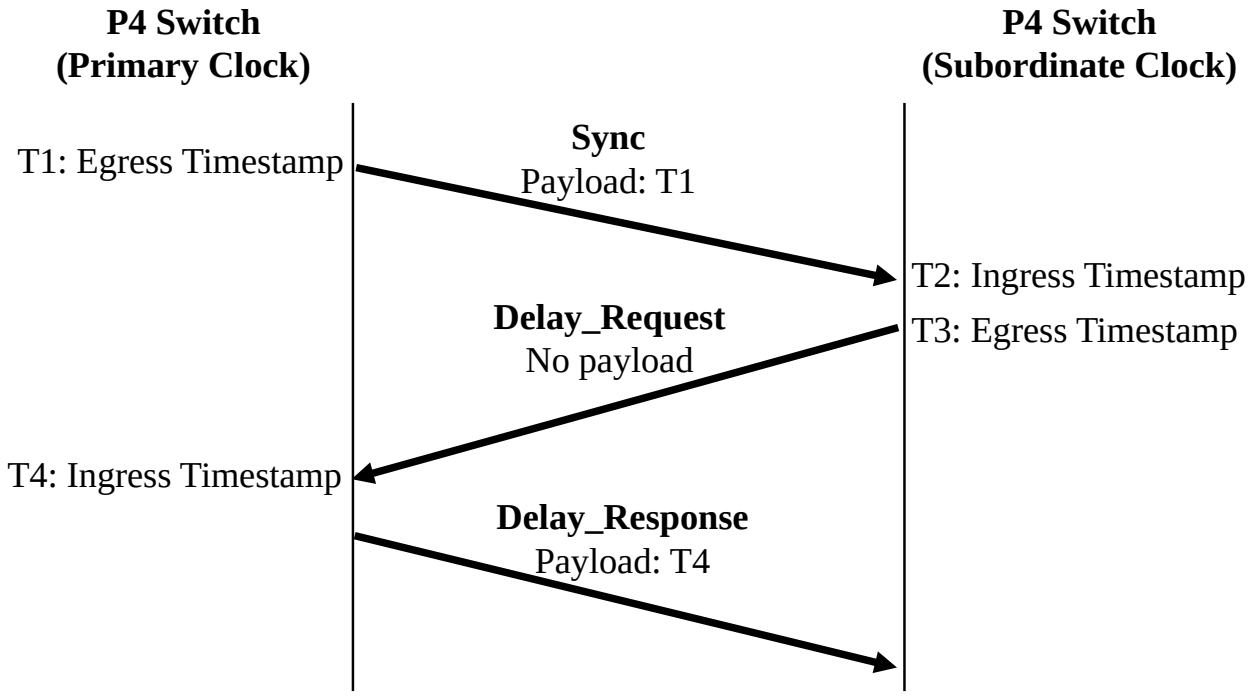


Figure 4.11 3-message PTP-1588 exchange sequence.

$$\frac{(T2 - T1) - (T4 - T3)}{2} = \text{clock\_offset}. \tag{4.2}$$

Our PTP implementation is a minimal proof of concept (POC) designed to demonstrate the P4 language's capability in implementing time synchronization protocols within the data plane.

We initially aimed to implement a 4-message exchange on the Behavioral Model version 2 (BMv2) software switch, a widely used tool for testing and developing P4 pipelines. Although BMv2 does not fully replace hardware implementations, it provides valuable insights into the behavior of P4-based network devices. It effectively simulates various real-world scenarios, allowing us to evaluate the performance of our eCPRI packet processing unit. However, the timestamps in this switch had issues that rendered them unsuitable for use in PTP-1588 synchronization. These issues include:

1. **Lack of necessary precision:** The standard ingress and egress timestamps in BMv2 are recorded at the start of the pipelines. However, accurate PTP clock difference calculations require capturing timestamps at the earliest and latest points in the packet

processing pipeline, ensuring that the calculated clock difference reflects the time spent by the packet within the switch.

BMv2 standard timestamps do not capture the precise time difference between actual packet ingress and the start of the P4 ingress pipeline, and they do not measure the time between actual packet egress and the start of the P4 egress pipeline. The execution time of the entire software switch pipeline is sensitive to the operating system's workload, which can lead to inconsistencies and variability in clock difference calculations. This underscores the importance of precise timestamp placement for accurate PTP-1588 synchronization.

2. **Measurement in microseconds:** PTP-1588 timestamps should be in nanoseconds, but the standard BMv2 timestamps are recorded in microseconds [35].

3. **Non-truncated format:** PTP-1588 uses 80-bit timestamp fields, where the 64 least significant bits represent the actual timestamp. As shown in Figure 4.13, these 64 bits are divided into two 32-bit sections: one for seconds and the other for nanoseconds [36]. However, BMv2 standard timestamps do not follow this truncated format.

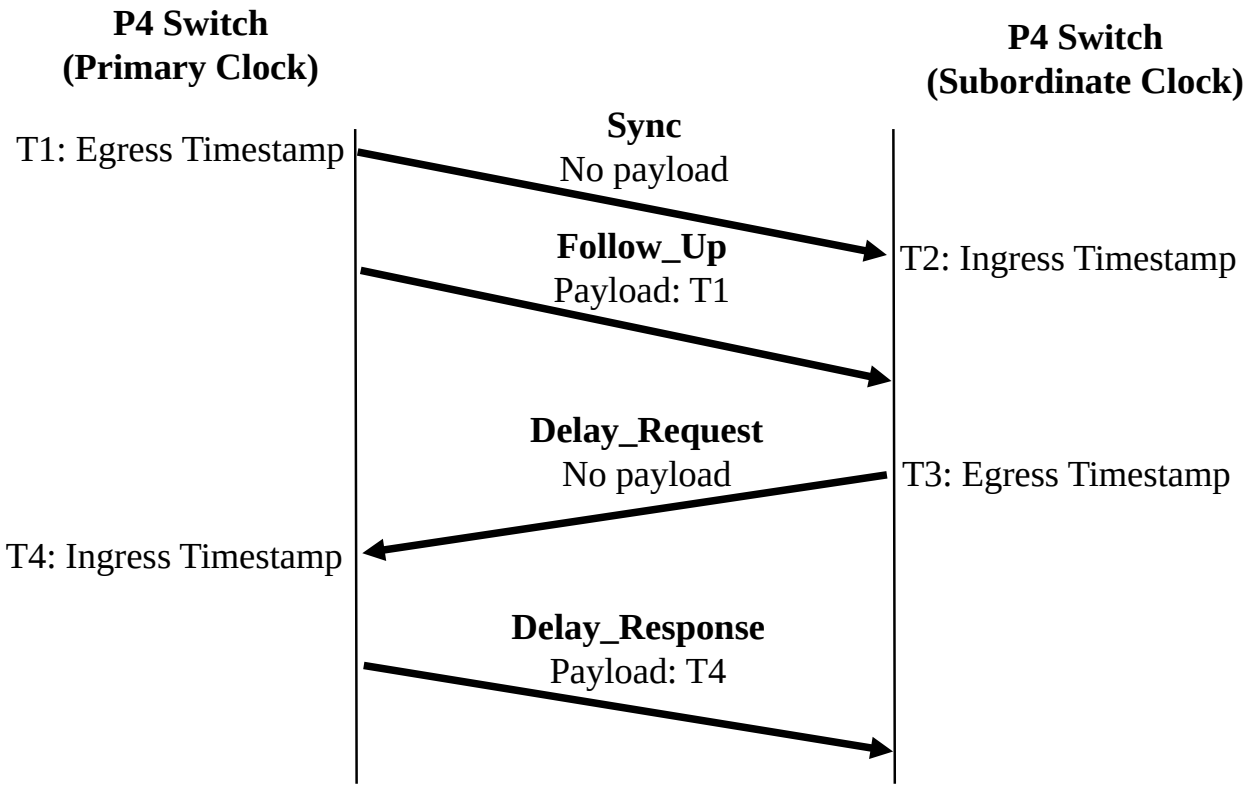


Figure 4.12 4-message PTP-1588 exchange sequence.

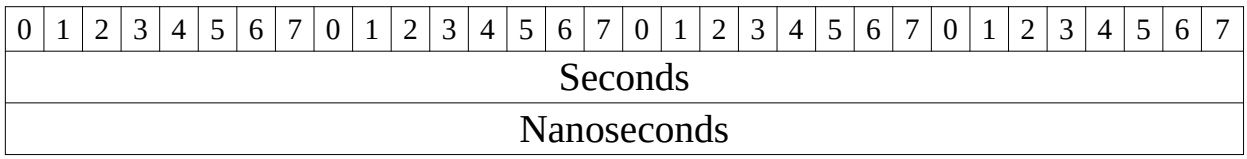


Figure 4.13 PTP-1588 Truncated Timestamp Format. [36]

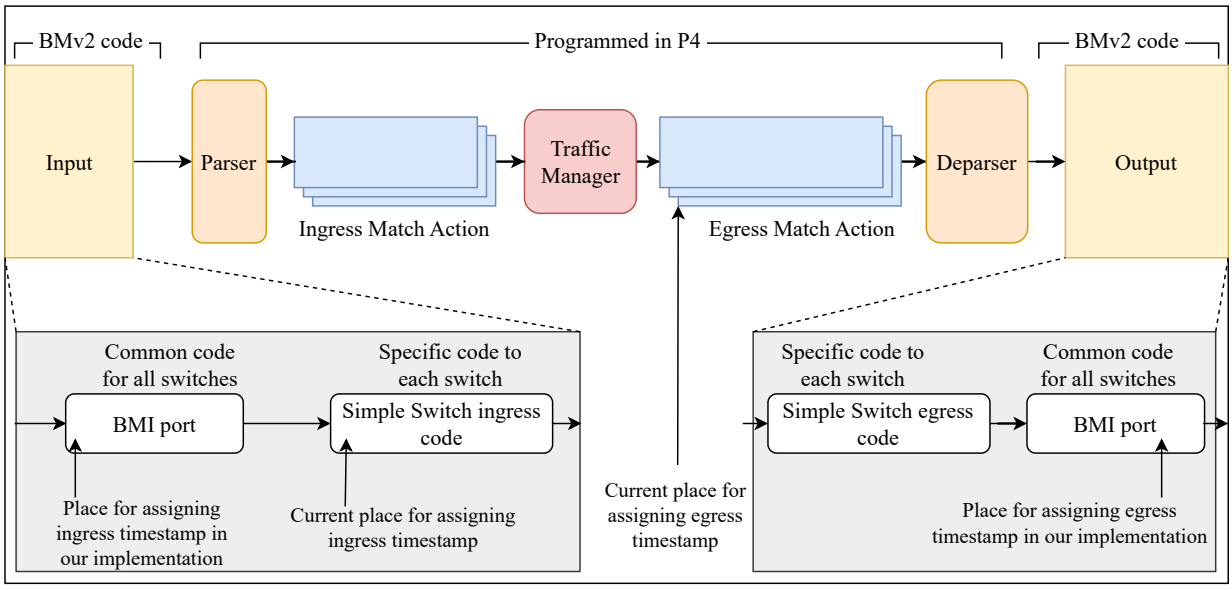


Figure 4.14 Locations in the pipeline where existing and proposed timestamps are measured.

4. **Lack of Unix Epoch format:** PTP-1588 timestamps should measure time in seconds since the Unix Epoch (January 1, 1970, at 00:00:00 UTC). In contrast, BMv2 software switches like simple_switch measure the time from when the software switch starts running.

To implement PTP-1588 in the BMv2 software environment, we added new timestamps to the BMv2 simple_switch, a widely used software switch within BMv2. These standardized timestamps can be used in our PTP-1588 implementation and other projects requiring precise timestamps.

We improved the precision of the ingress timestamp by measuring it at the earliest possible point, just after the software switch receives the packet, and before it enters the P4 pipeline. Since the P4 pipeline begins processing the packet after this point, the timestamp will be readily available within the pipeline.

Additionally, we improved the precision of the egress timestamp by capturing it at the latest

possible point, just before the packet exits the software switch. Because the packet reaches this point after the P4 pipeline has finished processing, this timestamp is unavailable within the P4 pipeline. To address this, the timestamp is automatically added to the packet after complete pipeline processing. New metadata fields were added to enable this timestamp and designate its position within the final packet. These fields are accessible within the P4 pipeline and facilitate a 3-message PTP exchange. Figure 4.14 illustrates the modifications made to the simple_switch behavioral model. The detailed implementation of this process is beyond the scope of this paper; however, a more comprehensive explanation, including slides and a recorded presentation, was shared at the P4 Workshop 2024 [47, 48].

4.4.6 Deparser and header demapper

When packets reach the deparser section of the P4 pipeline, final modifications are made based on specific instructions and metadata associated with each packet. At this stage, header fields are recombined with the payload, and additional headers are appended before the packet is sent to the output queue. The deparser ensures the packet is correctly formatted and ready for transmission according to network protocols.

4.5 Enhancing P4 flexibility through modular design

As previously mentioned, eCPRI's future-proof design includes provisions for vendor-specific message types, allowing new functionalities to be introduced over time. Table 4.1 lists the message type number allocations as specified in the eCPRI standard.

Without modularization, adding support for basic forwarding and conversion of a new packet type requires extensive code edits and additional code, making even a single new message type addition complex.

For instance, to process a specific eCPRI message type, such as type 5, the P4 program must first define the corresponding header type. This involves specifying the fields within the header, their data types, and their bit widths.

```
header ecpri5_t {
    bit<8> type5_measurement_id;
    bit<8> type5_action_type;
    bit<80> type5_timestamp;
    bit<64> type5_compensation_value;
}
```

Table 4.1 eCPRI Message Types [3].

Message Type #	**Name**
0	IQ Data
1	Bit Sequence
2	Real-Time Control Data
3	Generic Data Transfer
4	Remote Memory Access
5	One-way Delay Measurement
6	Remote Reset
7	Event Indication
8	IWF Start-Up
9	IWF Operation
10	IWF Mapping
11	IWF Delay Control
12 - 63	Reserved
64 - 255	Vendor Specific

Next, a header of this type must be defined to enable header extraction:

```
ecpri5_t  ecpri5;
```

Then, parser code must be written to handle this specific type:

```
packet.extract(hdr.ecpri);
if (hdr.ecpri.type == 5){
    packet.extract(hdr.ecpri5);
}
```

Additional code is required to implement functionalities such as sideband communication, concatenation, de-concatenation, and parsing. If this process is done manually, programmers must account for the unique requirements of each message type. For example, eCPRI message type 5, used for one-way delay measurement, should not be concatenated as this would introduce undesirable delays. However, the P4 processing pipeline must be capable of de-concatenating such messages if received within an Ethernet frame.

Furthermore, a new message type may require custom functionality. For instance, for eCPRI message type 5, we implemented an automatic system to respond to received messages:

```
if (hdr.ecpri.type == 5){
```

```
    bit<8> type5_measurement_id_var;
    type5_measurement_id_reg.read(
        type5_measurement_id_var, 0);
    if (hdr.ecpri5.type5_measurement_id
        != type5_measurement_id_var){
        standard_metadata.egress_spec =
            ethernet_port;
        macAddr_t macAddrChange =
            hdr.ethernet.dstAddr;
        hdr.ethernet.dstAddr =
            hdr.ethernet.srcAddr;
        hdr.ethernet.srcAddr =
            macAddrChange;
    }
}
```

Our design adopts a modular approach to simplify adding new message types. This involves defining the corresponding headers and implementing the necessary processing functions directly within the P4 code.

At the outset, we considered using O4 [34]. This domain-specific programming language allows custom packet handling in programmable networks, which converts to P4 and is designed to reduce the verbosity of P4 code. However, in the end, a macro preprocessor was chosen for this purpose.

4.5.1 O4

O4's `'for'` loop was initially appealing because it could simplify the addition of repetitive commands, like the ones shown below:

```
packet.emit(hdr.ecpri3);
packet.emit(hdr.ecpri4);
packet.emit(hdr.ecpri5);
packet.emit(hdr.ecpri6);
packet.emit(hdr.ecpri7);
packet.emit(hdr.ecpri8);
packet.emit(hdr.ecpri9);
packet.emit(hdr.ecpri10);
```

```
packet.emit(hdr.ecpri11);
```

This repetitive code could be implemented more concisely with O4's `'for/in'` loop primitive, which unrolls loops at compile time to fit within programmable devices' constraints. However, O4 only supports fixed-depth, compile-time-known loops, and single-dimensional iterators. These restrictions mean that each new data type or message type requires modifying existing code structures, hindering modularity. Due to these limitations, O4 was ultimately unsuitable for implementing a fully modular design.

4.5.2 Modularization using a macro preprocessor

Our implementation uses a macro preprocessor [49] to generate the necessary code for each message type, as it has far more capabilities than O4. This process only requires defining the name and size of the fields within a message header and specifying whether concatenation or de-concatenation should be performed. Originally intended to enhance existing C/C++ language preprocessors, we found that this macro preprocessor could be repurposed for our application.

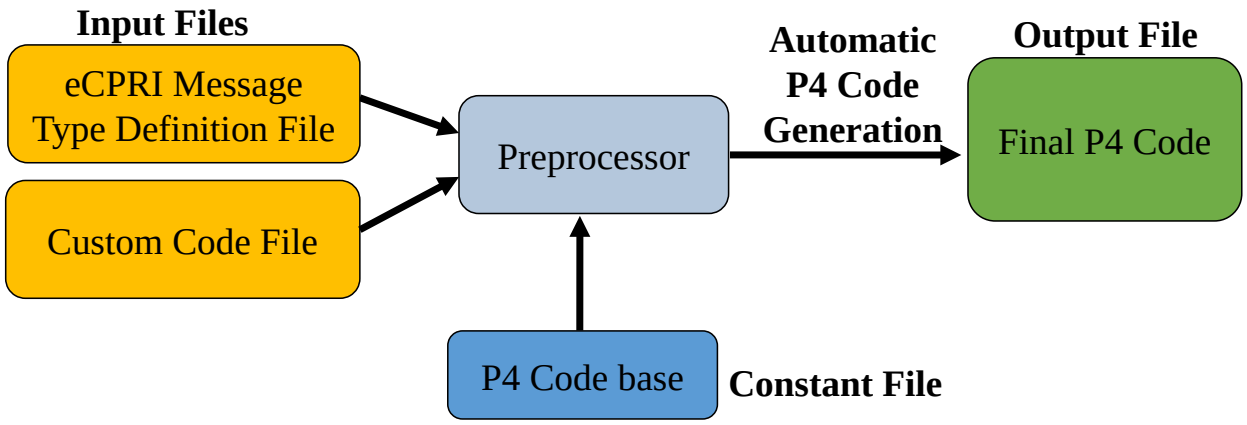


Figure 4.15 Modular pipeline generation by editing two definition files.

As shown in Figure 4.15, using this code generator preprocessor, the user only needs to include the new message types and custom code in two separate files and then run the preprocessor. The preprocessor will go through the pre-made base P4 code, add the necessary code according to the input files, and generate the entire P4 code with full support for the new message type. The code for existing message types was also created using this method. Below is an example of the modular addition of a message type with custom requirements for concatenation and de-concatenation using the `new_ecpri_message_type` keyword:

```
new_ecpri_message_type type_4 {
    type:4;
    total_size_in_bytes:12;
    no_concatenation_type;
    deconcatenation_type;
    field:mem_acc_id:8;
    field:op_type:8;
    field:element_id:16;
    field:address:48;
    field:length:16;
};
```

Additionally, users can add custom code in a separate file by indicating placeholder names within the P4 code base using the `custom_code` keyword:

```
custom_code
place_marker_ingress_cloning_port_end{
    hdr.ecpri9.setInvalid();
}
```

To process definitions of new message types, the macro "`new_ecpri_message_type`" must first be defined:

```
@define new_ecpri_message_type {
...
}
```

Within this macro definition, the behavior of the preprocessor is specified.

The macro preprocessor operates on the principle of keyword-triggered pattern matching and token substitution [49]. A macro is defined using the "`@define`" directive. For example:

```
@define new_macro {
    ( pattern_a ) => ( test1 )
    ( pattern b ) => ( test2 )
    ...
}
```

When the keyword "`new_macro`" is encountered, the preprocessor matches the subsequent token stream against the patterns defined in the macro body. If a match is found, the corresponding right-hand token stream replaces the keyword and matching tokens. For instance, "`new_macropattern_a`" would be replaced with "`test1`."

Additionally, a loop mechanism can handle long token streams, enabling repetitive pattern-matching:

```
@define new_macro {
    ...
    @for( $p : $pattern1)(
        @match ($p){
            ( pattern_a ) => ( test1 )
            ( pattern b ) => ( test2 )
            ...
        }
    )
    ...
}
```

We use these mechanisms to create token lists that generate the necessary code for each section. For instance, if the preprocessor detects the keyword "`concatenation`," it generates a list used to create the code supporting the concatenation process. The pseudocode below illustrates the creation of such lists:

```
@define new_macro {
  ...
  @for( $p : $pattern1)(
    @match ($p){
      ...
      (concatenation_type)=>
          (@push_back $concat_list ($p))
      ...
    }
  )
  ...
}
```

Table 4.2 Modularity Metrics: Standard P4 vs. Preprocessor.

Metric	**Standard P4**	**Preprocessor**	**Improvement**
LoC per Type	40–80	5–12	85% less
Time to Add Type	1–4 hrs	10–60 min	5x faster
Complexity	High	Low	Simplified

Preprocessor code can be inserted anywhere in the P4 pipeline where support for new message types is required. This code extracts the token stream from the list and generates the corresponding P4 code for any message type present in the list.

As a simple example, consider the following preprocessor code:

```
@for( $a : $type_list )(
    ecpri$a@unquote "_t \tecpri"$a;
)
```

This preprocessor code generates the following definitions:

```
ecpri3_t  ecpri3;
ecpri4_t  ecpri4;
ecpri5_t  ecpri5;
ecpri6_t  ecpri6;
ecpri7_t  ecpri7;
ecpri8_t  ecpri8;
ecpri9_t  ecpri9;
ecpri10_t  ecpri10;
ecpri11_t  ecpri11;
```

Table 4.2 compares modularity metrics between the standard P4 implementation and the preprocessor-based approach. The improvements in modularity significantly reduce the complexity of adding new eCPRI message types. Specifically, the preprocessor reduces the number of lines of code (LoC) per message type by up to 85%, minimizes the time required to implement a new type, and simplifies the overall complexity. This modular design decreases the risk of human error and accelerates the development process.

4.6 Testing the eCPRI conversion unit

Comprehensive testing is essential for ensuring the reliability and functionality of network packet processing units. This involves generating diverse test packets to simulate various network scenarios. Given the complexity of our eCPRI conversion unit, which comprises multiple subsystems, a multi-layered testing approach is necessary.

4.6.1 Testing packet conversion functionality

We thoroughly tested the eCPRI conversion unit code to ensure its reliability and security. We successfully identified and resolved potential errors by subjecting it to a wide range of packet scenarios. Additionally, we verified that the code processes packets correctly, accurately extracts and interprets data, and takes appropriate actions based on the packet content.

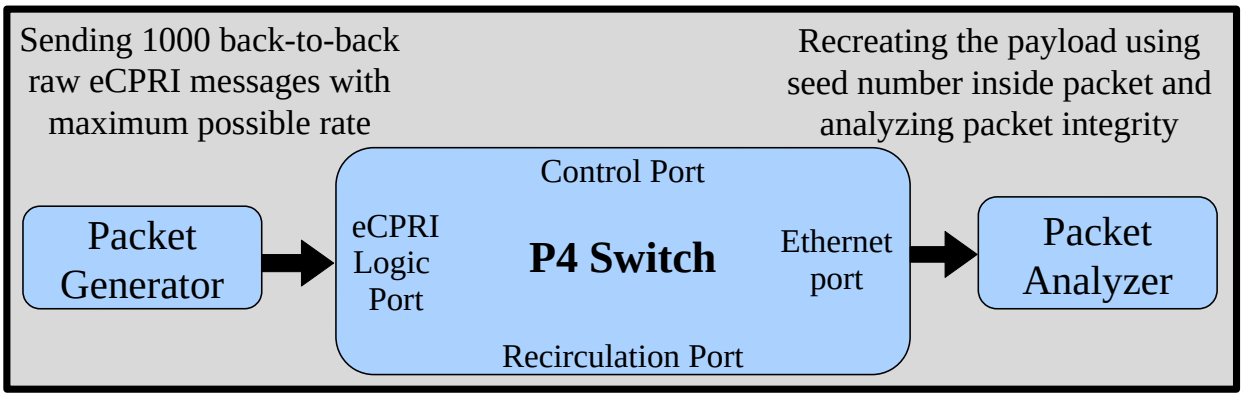


Figure 4.16 Setup for testing packet conversion functionality.

Two custom Python scripts were used to test the basic header conversion and concatenation/de-concatenation functionality, as shown in Figure 4.16. The first script generates randomized test packets with varying lengths, types, and data payloads. Each packet uses a unique seed value to ensure consistent randomization across different test runs. While specific fields, such as packet length, must adhere to predefined values, other fields are randomized to simulate diverse network scenarios. This approach allowed us to thoroughly evaluate the unit's performance under various conditions.

The second script functions as a packet analyzer, capturing and verifying the output generated by the eCPRI unit. Upon receiving a packet, the receiver recreates the original byte string using the embedded seed value. This reconstructed string is then compared to the received string, allowing for the detection of discrepancies. Additionally, the receiver maintains a packet count to verify that all packets are forwarded correctly and no packets are lost.

To ensure the final eCPRI packets are correctly formed, packets were intercepted using Wireshark software. Wireshark can detect anomalies such as incorrect header sizes or fields and mark the packet as broken. Any header field with incorrect values is also flagged. Some header fields, such as packet size, are essential for concatenation and de-concatenation processes. To enhance resilience, the P4 pipeline drops any malformed packets. Furthermore, critical fields, such as packet length, are inspected. The conversion unit also examines incoming Ethernet packets to ensure their types are supported and filters out any Ethernet packet that does not contain an eCPRI, PTP, or IP packet.

4.6.2 Testing the PTP-1588 synchronization implementation

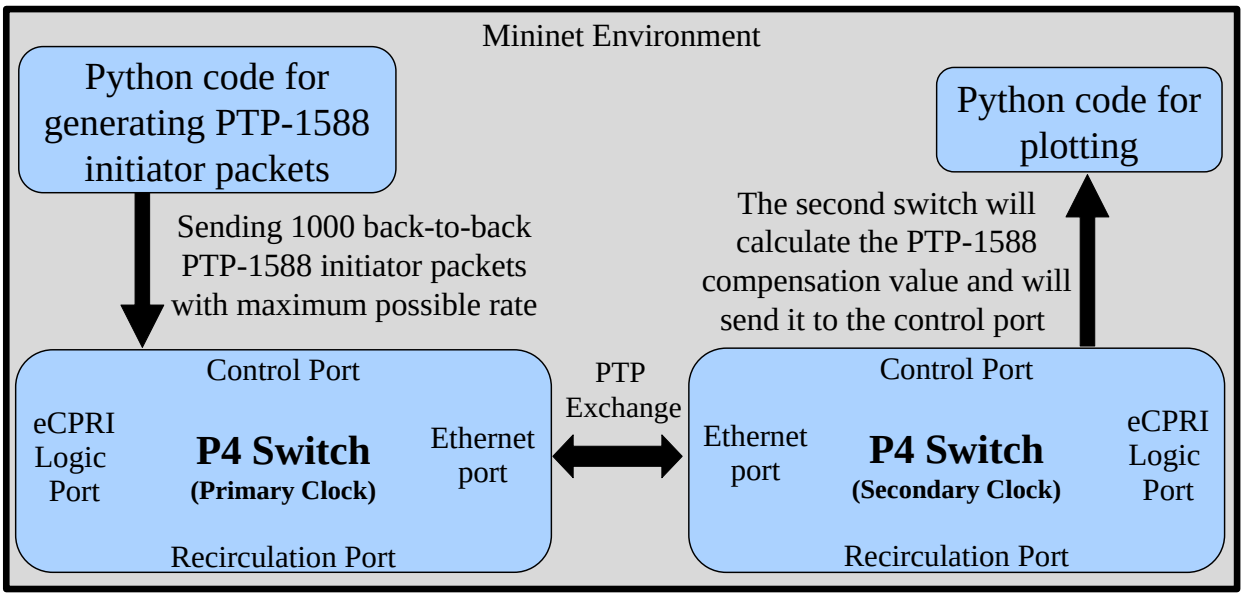


Figure 4.17 Setup for testing the synchronization unit in the Mininet environment.

To evaluate PTP-1588 synchronization, we utilized the Mininet network emulation platform in the setup shown in Figure 4.17. Our testing encountered challenges due to the Linux network stack's default behavior of broadcasting packets like Multicast Domain Name System (mDNS) and Internet Control Message Protocol version 6 (ICMPv6) to all active ports. The logic port is intended to connect to hardware that generates eCPRI payloads; messages sent to this port do not have headers, as they are not packets but data streams. We used virtual Ethernet ports to simulate this connection in a software environment. However, mDNS and ICMPv6 packets could inadvertently reach the logic port, causing the switch to process them as valid eCPRI payloads and potentially leading to unintended eCPRI transmissions and recirculations.

As a workaround, eCPRI processing was temporarily disabled during PTP-1588 testing. Since eCPRI and PTP-1588 over Ethernet operate independently, this adjustment did not invalidate the testing process.

To initiate PTP-1588 synchronization, the test script sends a PTP initiator packet to the control port, simulating a command from the control plane. This triggers a three-message PTP exchange within the pipeline, as depicted in Figure 11. Upon receiving the third PTP message, the subordinate switch calculates the necessary compensation value. To access this value for analysis and plotting, the subordinate switch encapsulated it within a packet and transmitted it to its control port.

Extensive testing was conducted by sending 1,000 initiator packets to the control port of the primary switch at the maximum possible rate. We compared the performance of our implementation using both standard timestamps and our proposed, more precise timestamps. The results significantly improved PTP-1588 precision when using the new ingress timestamp and automatic egress timestamp.

Some jitter was observed during testing, with a small percentage of measurements showing higher errors than the general trend. This variability was attributed to fluctuations in the Linux network stack or the system's overall workload. Additional tests on the new timestamps alone revealed that background processes and the operating system caused part of the final error.

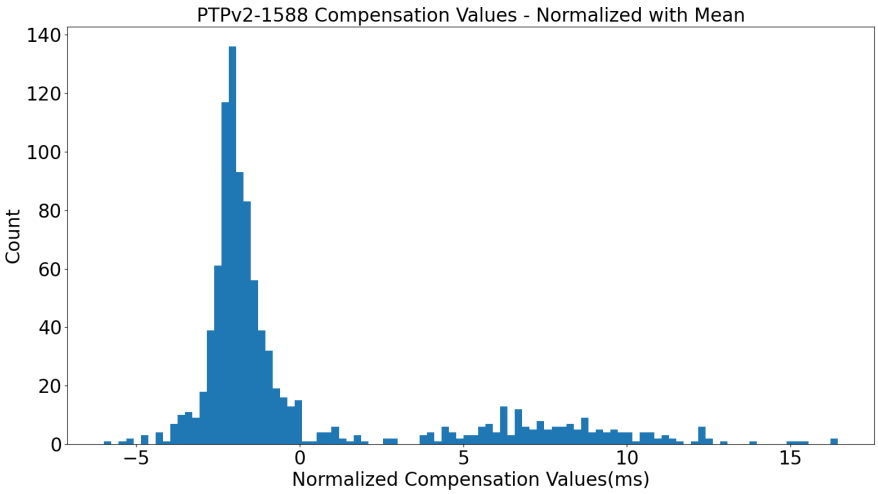
By reducing processing loads and network traffic, we mitigated the occurrence of these abnormally high error values. While it's possible that these outliers could be further reduced by disabling other system processes, this was beyond the scope of our project.

To better assess the impact of our new timestamps, we conducted a detailed analysis, comparing the overall distribution and the most precise 90% of measurements. By excluding the 10% of results with the highest errors, we could isolate the effects of our new timestamps and reduce the influence of jitter and other external factors.

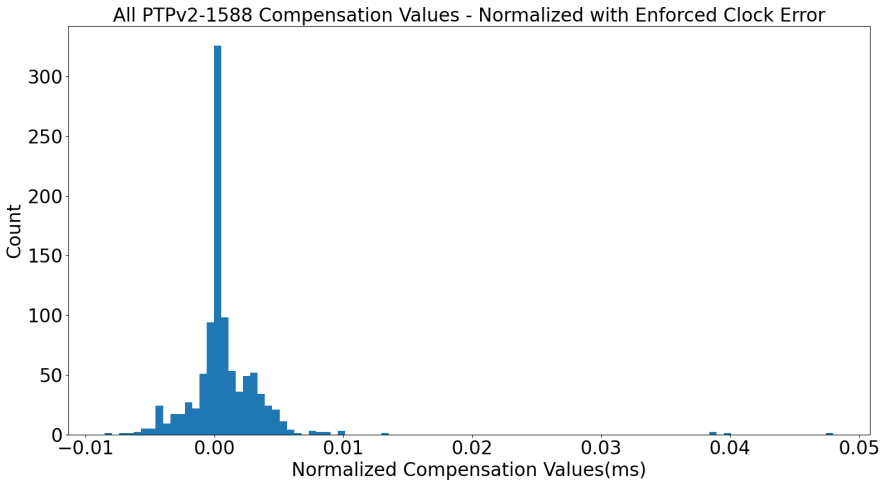
Comparison of error distributions with standard and new timestamps

Figures 4.18a and 4.18b show the final error values for the 1000 PTP-1588 tests using the standard timestamps and the new, precise timestamps.

Calculating error values is not straightforward when using the standard timestamps provided in the simple_switch software. These timestamps measure the time since the software switch started, which is challenging to coordinate. As a result, determining the actual time difference between the two switches is challenging. To normalize the values and calculate error rates,



(a) Using standard timestamps, error values are shown in tens of milliseconds.



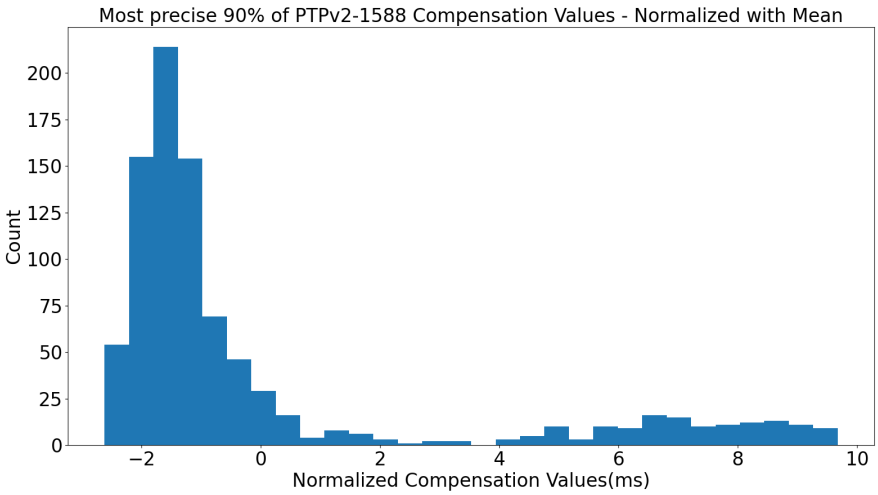
(b) Using precise ingress and automatic egress timestamps, reducing error values to tens of microseconds.

Figure 4.18 Error distributions for PTP-1588 synchronization using different timestamps.

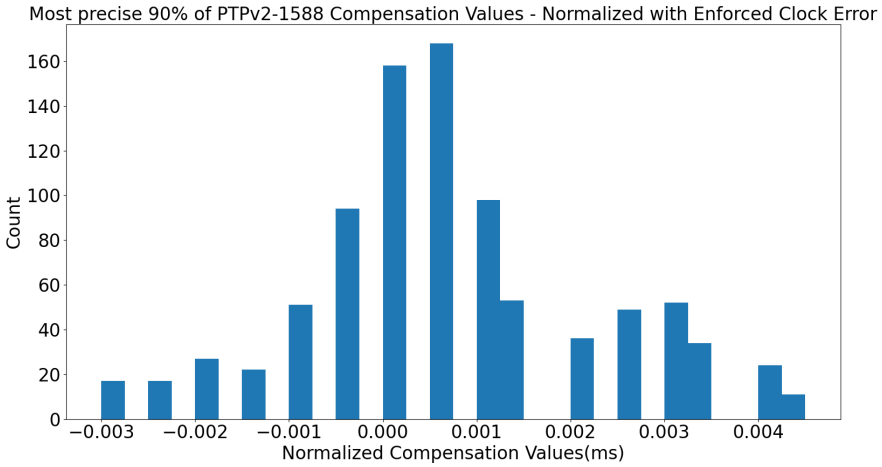
we used the average of the calculated time differences across all 1000 tests.

In contrast, calculating error values with the new timestamps was more straightforward due to the added capability of enforcing artificial clock errors. By including this artificial error as an input parameter in the software switch, we could assess the effects of clock discrepancies on any communication protocol.

Figures 4.19a and 4.19b show the final error values distribution for the 1000 PTP-1588 tests using standard and new timestamps, excluding the 10% most erroneous values. By focusing



(a) Using standard timestamps, normalized with mean values.



(b) Using new timestamps, normalized with enforced clock error.

Figure 4.19 Distribution of the most precise 90% of PTP-1588 compensation values.

on the 90th percentile, we can mitigate the effects of system jitter and highlight the differences in precision between the standard and new timestamps.

As shown in Figure 4.19b, the compensation values are significantly more concentrated with the new timestamp implementation, with error values reduced to a few microseconds. In contrast, the distribution using standard timestamps in Figure 4.19a shows a broader range of error values, spanning tens of microseconds, indicating lower precision.

Table 4.3 summarizes the results of two PTP-1588 implementations: one using the standard BMv2-provided timestamps and the other employing our newly proposed timestamping

Table 4.3 Comparison of PTP-1588 Error Ranges with Standard BMv2 and New Timestamps

Metric	**BMv2 (μs)**	**New (μs)**	**Improvement (%)**
All Values	24,000	60	99.75
90th Percentile	13,000	9	99.93

mechanism.

When considering all values, the new timestamping system reduces the error range by 99.75%, from 24 milliseconds to 60 microseconds. Further analysis of the 90th percentile of results reveals that the new timestamping mechanism achieves a 99.93% reduction in error, lowering the range from 13 milliseconds to 9 microseconds. The 90th percentile precision improvement ensures that most packets experience minimal timing errors. These significant reductions underscore our proposed timestamping solution's enhanced precision and reliability.

This dramatic improvement in timing precision directly impacts the performance of eCPRI fronthaul networks, enabling more accurate synchronization between Baseband Units (BBUs) and Remote Radio Heads (RRHs). By reducing timing errors to the microsecond level, our implementation ensures compatibility with 5G's stringent timing requirements, enhancing both network operations' reliability and efficiency.

4.7 Discussion on P4's limitations and extensions for complex protocol processing

P4 is primarily designed for low-latency packet processing pipelines, which introduces several challenges when implementing complex network functionalities such as the eCPRI conversion unit.

One of P4's widely acknowledged limitations is its inability to manipulate packet payloads directly. While this design choice aligns with P4's core philosophy of describing packet forwarding behavior rather than data manipulation, it becomes problematic in scenarios like concatenation. Packet concatenation, which combines multiple smaller packets into a single Ethernet frame, is essential in many network protocols and hardware implementations. Without this capability, P4-based systems risk incompatibility with non-P4 implementations that support concatenation and de-concatenation, thereby limiting interoperability in heterogeneous network environments.

Additionally, P4's handling of variable-length headers is restrictive. While these headers are ideal for extracting payloads, the current specification only allows parsing and deparsing with-

out further processing [2]. Extending P4 to support direct manipulation of variable-length headers could simplify payload handling, but such a change might introduce complexities that challenge the language's hardware implementation and efficiency.

A practical alternative is to enhance P4's support for extern objects, enabling them to process entire packets or variable-length headers. Current P4 specifications do not permit externs to accept complete packets or variable-length headers, necessitating workarounds such as dynamic extraction using headers of varying sizes. While effective, these methods increase implementation complexity and highlight the need for more flexible packet-processing primitives within the P4 language.

Beyond extraction, storing packet payloads also poses challenges. The lack of support for variable-length registers or multi-dimensional register arrays limits the ability to store payloads efficiently. Enhancing P4 to support such capabilities could facilitate operations like concatenation and de-concatenation without relying on externs. For example, enabling iterative access to multi-dimensional registers could allow P4 pipelines to store payloads from multiple packets using constant-size headers, significantly improving modularity and flexibility.

While these limitations reflect the current state of P4, they also point to opportunities for future enhancements that could extend the language's applicability to more complex and modular network protocols. By addressing these challenges, P4 could evolve into a more comprehensive tool for modern programmable networks.

4.8 Conclusion

In this research work, we implemented a flexible and efficient eCPRI protocol processing unit using the P4 language, tailored for 5G fronthaul applications. By addressing the limitations of P4 in handling packet payloads, we proposed novel techniques for packet concatenation and de-concatenation and integrated a macro preprocessor to enhance modularity. This modular approach enables adding new eCPRI message types and custom processing with minimal code changes. Our approach reduced lines of code (LoC) per type by 85%, decreased configuration time by up to 5x, and simplified complexity compared to traditional P4 workflows.

Additionally, we integrated PTP-1588 support into the unit, overcoming precision limitations in BMv2's standard timestamps by introducing new precise ingress and automatic egress timestamps. These enhancements reduced error margins from 24,000 microseconds to just 60 microseconds (99.75% improvement), bringing the 90th percentile error down from 13,000 microseconds to 9 microseconds (99.93% improvement). Artificial clock error functionality

was also added to the BMv2 environment, streamlining the testing of time-sensitive protocols.

These contributions advance the development of programmable network devices, demonstrating how P4 can be a more modular, adaptable, and high-precision language for innovations in 5G fronthaul and other time-sensitive applications.

CHAPTER 5 CONCLUSION AND FUTURE WORKS

5.1 Summary of Works

We designed a novel eCPRI over Ethernet packet processing pipeline using P4 language, incorporating advanced network functionalities such as packet concatenation, de-concatenation, and PTP-1588 synchronization. The P4 language's limited functionality, designed to describe low-latency packet processing pipelines, created challenges in our pursuit of building our eCPRI conversion unit. We overcame these issues by utilizing externs to create a more customized solution.

Since the eCPRI is not a fully stable protocol with much room for improvement and the possibility of adding new messages in the future, a modular design was adopted to accommodate these possible scenarios and ease the process of customizing the unit.

While our implementation currently concatenates packets up to a maximum size of 511 bytes, it is important to note that this limitation does not necessarily hinder its effectiveness. Smaller packets, with their relatively larger header sizes, experience a more significant reduction in overhead when concatenated. This makes concatenating smaller packets particularly beneficial in terms of improving network efficiency.

Furthermore, existing software switches did not offer precise timestamps but only non-standard and inaccurate mechanisms, leaving us unable to test the final design methodically. We started editing the existing software switch and increased timestamp accuracy. Moreover, we added an automatic egress timestamp unit to the simple switch software, which enabled hardware timestamping used in the PTP-1588 synchronization protocol.

Our PTP-1588 implementation, dramatically improved the precision of the synchronization. Using our new timestamps, we reduced the synchronization error from 24,000 microseconds to only 60 microseconds, displaying a 99.75% improvement.

Additionally, we incorporated modularity into our design by using a preprocessor to create P4 code for new eCRPI header types, decreasing definition time by up to 5x. This enhancement ensured the forward compatibility of our P4 pipeline.

5.2 Limitations

P4 excels at high-speed packet processing. However, implementing complex functionalities like the eCPRI conversion unit might require careful design considerations. P4's main ob-

jective is to describe packet forwarding behavior rather than data manipulation, and this limitation will create issues in scenarios like concatenation.

While variable-length headers are ideal for payload extraction, the current specification only permits parsing and deparsing using these headers, precluding any further manipulation [2]. Using P4 externs to allow direct packet manipulation could simplify payload processing. However, the current P4 specification [2] does not support the transfer of packets from the P4 pipeline to externs. Efficient payload storage is also challenging due to P4's lack of variable-length registers or multi-dimensional register arrays.

Besides the limitations of the P4 language for our use case, our implementation also has shortcomings. A final P4 product must be robust against malformed and erroneous packets and process unwanted packets appropriately. While we have addressed many of these issues, there is still room for improvement under real-world network conditions. Moreover, our experiments did not include creating concatenated packets for multiple simultaneous streams.

The preprocessor for automatic P4 code generation provides no error reporting, complicating debugging. The preprocessor only allows custom P4 code insertion at designated placeholders, preventing arbitrary line insertion. A further problem with the preprocessor is its tendency to insert extra newlines.

5.3 Future Research

While this research has made significant strides in exploring the potential of P4 for eCPRI packet processing, there are several avenues for future exploration:

- Translating the proposed P4-based design into a hardware implementation using an FPGA would enable high-performance, low-latency eCPRI packet processing. This would provide valuable insights into the real-world performance and limitations of the proposed approach.
- Further investigation into advanced P4 features, such as stateful programming and event-driven packet processing, could unlock new applications for the P4 language.
- More extensive work can be done to ensure the safety of eCPRI packet processing pipelines. Implementing robust security measures can be a decisive advantage for P4 over other approaches for designing packet processing pipelines, as P4 is one of the most adaptable and customizable solutions in this field.

REFERENCES

[1] P. Bosshart *et al.*, "P4: programming protocol-independent packet processors," *SIGCOMM Comput. Commun. Rev.*, vol. 44, no. 3, p. 87–95, Jul. 2014. [Online]. Available: https://doi.org/10.1145/2656877.2656890

[2] P4 Language Consortium, "$P4_{16}$ language specification," P4.org, Tech. Rep., Nov. 2021, accessed: 30-Aug-2024. [Online]. Available: https://p4.org/p4-spec/docs/P4-16-v1.2.3.html

[3] C. cooperation, "Common public radio interface: ecpri interface specification v2.0," CPRI cooperation, Tech. Rep., May 2019. [Online]. Available: https://www.cpri.info/downloads/eCPRI_v_2.0_2019_05_10c.pdf

[4] *eCPRI Intel® FPGA IP User Guide*, Intel Corporation, August 2024, version 24.2-3.0.3. [Online]. Available: https://cdrdv2.intel.com/v1/dl/getContent/827767?fileName=ug-683685-827767.pdf

[5] C. cooperation, "Common public radio interface," 2003. [Online]. Available: http://www.cpri.info/press_20030930.html

[6] A. Saadani *et al.*, "Digital radio over fiber for lte-advanced: Opportunities and challenges," in *2013 17th International Conference on Optical Networking Design and Modeling (ONDM)*, 2013, pp. 194–199.

[7] J. Gomes, J. A. L. Silva, and M. E. V. Segatto, "Reducing the 5g fronthaul traffic with o-ran," in *2019 SBMO/IEEE MTT-S International Microwave and Optoelectronics Conference (IMOC)*, 2019, pp. 1–3.

[8] O.-R. Alliance, "O-RAN: Towards an Open and Smart RAN," Oct. 2018. [Online]. Available: https://assets-global.website-files.com/60b4ffd4ca081979751b5ed2/60e5afb502810a0947b3b9d0_O-RAN%2BWP%2BFInal%2B181017.pdf

[9] C. cooperation, "Common public radio interface: Cpri specification," CPRI cooperation, Tech. Rep., Oct. 2015. [Online]. Available: http://www.cpri.info/downloads/CPRI_v_7_0_2015-10-09.pdf

[10] I. . W. Group, "Ieee 1914 working group - p1914.1," 2023. [Online]. Available: https://sagroups.ieee.org/1914/p1914-1/

[11] P. L. Consortium, "P4 language consortium," 2016. [Online]. Available: https://p4.org/

[12] ——, "p4lang/behavioral-model: the reference p4 software switch," GitHub. [Online]. Available: https://github.com/p4lang/behavioral-model

[13] ——, "P4 16 reference compiler," GitHub, 2024. [Online]. Available: https://github.com/p4lang/p4c

[14] P. Kartaschoff, "Synchronization in digital communications networks," *Proceedings of the IEEE*, vol. 79, no. 7, p. 1019–1028, Aug 1991.

[15] M. Lévesque and D. Tipper, "A survey of clock synchronization over packet-switched networks," *IEEE Communications Surveys & Tutorials*, vol. 18, no. 4, pp. 2926–2947, 2016.

[16] X. Zhang, H.-G. Ryu, and J. up Kim, "Suppression of synchronization errors in ofdm based carrier aggregation system," in *2010 16th Asia-Pacific Conference on Communications (APCC)*, 2010, pp. 106–111.

[17] H. Li *et al.*, "Analysis of the synchronization requirements of 5g and corresponding solutions," *IEEE Communications Standards Magazine*, vol. 1, no. 1, pp. 52–58, 2017.

[18] H. Kwak *et al.*, "Dynamic maintenance method of timing synchronization for soft handover," in *2009 11th International Conference on Advanced Communication Technology*, vol. 01, 2009, pp. 428–431.

[19] D. Lee *et al.*, "Coordinated multipoint transmission and reception in lte-advanced: deployment scenarios and operational challenges," *IEEE Communications Magazine*, vol. 50, no. 2, pp. 148–155, 2012.

[20] A. Kaloxylos, A. Gavras, and R. De Peppe, "Empowering Vertical Industries through 5G Networks - Current Status and Future Trends," Sep. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3698113

[21] M. Koivisto *et al.*, "Joint device positioning and clock synchronization in 5g ultra-dense networks," *IEEE Transactions on Wireless Communications*, vol. 16, no. 5, pp. 2866–2881, 2017.

[22] I. Freire *et al.*, "Testbed evaluation of distributed radio timing alignment over ethernet fronthaul networks," *IEEE Access*, vol. 8, pp. 87 960–87 977, 2020.

[23] D. T. Kiet *et al.*, "Research and implementation of ecpri processing module for fronthaul network on fpga in 5g – nr gnodeb base station," in *2020 4th International Conference on Recent Advances in Signal Processing, Telecommunications & Computing (SigTelCom)*, 2020, pp. 1–5.

[24] R. M. Rao, M. Fontaine, and R. Veisllari, "A reconfigurable architecture for packet based 5g transport networks," in *2018 IEEE 5G World Forum (5GWF)*, 2018, pp. 474–477.

[25] S. T. Le *et al.*, "400gb/s real-time transmission supporting cpri and ecpri traffic for hybrid lte-5g networks," in *2020 Optical Fiber Communications Conference and Exhibition (OFC)*, 2020, pp. 1–3.

[26] ——, "100gbps dmt asic for hybrid lte-5g mobile fronthaul networks," *Journal of Lightwave Technology*, vol. 39, no. 3, pp. 801–812, 2021.

[27] A. AlSabeh *et al.*, "P4ddpi: Securing p4-programmable data plane networks via dns deep packet inspection," *Proceedings 2022 Workshop on Measurements, Attacks, and Defenses for the Web*, 2022.

[28] A. Alsabeh *et al.*, "Effective dga family classification using a hybrid shallow and deep packet inspection technique on p4 programmable switches," in *ICC 2023 - IEEE International Conference on Communications*, 02 2023.

[29] A. AlSabeh *et al.*, "On dga detection and classification using p4 programmable switches," *Computers & Security*, vol. 145, p. 104007, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167404824003122

[30] S. Gupta *et al.*, "Deep4r: Deep packet inspection in p4 using packet recirculation," in *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*, 2023, pp. 1–10.

[31] A. Ahad *et al.*, "Dpidns:a deep packet inspection based ips for security of p4 network data plane," in *2023 International Conference on Smart Computing and Application (ICSCA)*, 2023, pp. 1–8.

[32] P. G. Kannan, R. Joshi, and M. C. Chan, "Precise time-synchronization in the data-plane using programmable switching asics," in *Proceedings of the 2019 ACM Symposium on SDN Research*, ser. SOSR '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 8–20. [Online]. Available: https://doi.org/10.1145/3314148.3314353

[33] H. Soni *et al.*, "Composing dataplane programs with μp4," in *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication*, ser. SIGCOMM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 329–343. [Online]. Available: https://doi.org/10.1145/3387514.3405872

[34] A. G. Alcoz *et al.*, "Reducing p4 language's voluminosity using higher-level constructs," in *Proceedings of the 5th International Workshop on P4 in Europe*, ser. EuroP4 '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 19–25. [Online]. Available: https://doi.org/10.1145/3565475.3569078

[35] *IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems*, IEEE Std., 2008.

[36] T. Mizrahi, J. Fabini, and A. Morton, "Guidelines for Defining Packet Timestamps," RFC 8877, Sep. 2020. [Online]. Available: https://www.rfc-editor.org/info/rfc8877

[37] I. A. Advanced Micro Devices, "Amd petalinux tools," Aug 2024. [Online]. Available: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/embedded-software/petalinux-sdk.html

[38] N. Budhdev *et al.*, "Fsa: fronthaul slicing architecture for 5g using dataplane programmable switches," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 723–735. [Online]. Available: https://doi.org/10.1145/3447993.3483247

[39] R. Ricart-Sanchez *et al.*, "P4-netfpga-based network slicing solution for 5g mec architectures," in *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2019, pp. 1–2.

[40] R. Doriguzzi-Corin *et al.*, "Introducing packet-level analysis in programmable data planes to advance network intrusion detection," *Computer Networks*, vol. 239, p. 110162, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1389128623006072

[41] S.-Y. Wang *et al.*, "High-speed data-plane packet aggregation and disaggregation by p4 switches," *Journal of Network and Computer Applications*, vol. 142, pp. 98–110, 2019. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804519301687

[42] S.-Y. Wang, J.-Y. Li, and Y.-B. Lin, "Aggregating and disaggregating packets with various sizes of payload in p4 switches at 100 gbps line rate," *Journal of Network and Computer Applications*, vol. 165, p. 102676, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1084804520301508

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

[44] M. P. Contributors, "Mininet: An instant virtual network on your laptop (or other pc) - mininet," 2022. [Online]. Available: https://mininet.org/

[45] J. Tourrilhes, "Packet frame grouping: improving ip multimedia performance over csma/ca," in *ICUPC '98. IEEE 1998 International Conference on Universal Personal Communications. Conference Proceedings (Cat. No.98TH8384)*, vol. 2, 1998, pp. 1345–1349 vol.2.

[46] D. Kliazovich and F. Granelli, "On packet concatenation with qos support for wireless local area networks," in *IEEE International Conference on Communications, 2005. ICC 2005. 2005*, vol. 2, 2005, pp. 1395–1399 Vol. 2.

[47] B. Pontikakis and F.-R. Boyer, "Supporting ptp-1588 in bmv2: A proposed ingress and egress timestamping scheme," 2024. [Online]. Available: https://p4.org/wp-content/uploads/2024/10/5-2024-P4-Workshop-Boyer_Pontikakis.pdf

[48] ——, "Supporting ptp-1588 in bmv2 - bill pontikakis and francois-raymond boyer (polytechnique montreal)," YouTube, Oct 2024. [Online]. Available: https://www.youtube.com/watch?v=wD6FVG3lavo

[49] M. Soloviev, "Github - blackhole89/macros: A more powerful c/c++ preprocessor." GitHub. [Online]. Available: https://github.com/blackhole89/macros