

Combining High-Level-Synthesis and Register-Transfer Level Design for Programmable Hardware

by

Kimiya Mohammadtaheri

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Kimiya Mohammadtaheri 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

As network speeds continue to grow to hundreds of Gbps and beyond, offloading transport-layer functionality to programmable Network Interface Cards (NICs) has gained traction for improving performance and enabling higher-level offloads. FPGAs on NICs offer low latency and high throughput but are notoriously difficult to program. To reduce developer effort for hardware transport, prior works propose hardware architectures with reusable fixed-function modules for common transport data structures and operations, and programmable modules for protocol-specific operations. However, they either still require intricate Verilog programming for their programmable modules or are tied to specific protocols or pipeline abstractions.

In this work, we explore the design of a programmable hardware architecture for transport, called HTraP, that (1) offers a simple, intuitive programming interface without requiring detailed understanding of the rest of the architecture for effective programming, and (2) supports a broad range of transport protocols with varying levels of complexity. HTraP has one main programmable module that can be programmed to capture the core protocol logic in simple C code that is amenable to automated, efficient hardware generation with HLS. The generated hardware can then plug into the rest of HTraP which implements complex transport operations through a protocol-agnostic interface.

Acknowledgements

I would like to thank my supervisor, Mina Tahmasbi Arashloo, for her guidance, encouragement, and invaluable support throughout the course of my MMath program. I thoroughly enjoyed every minute we spent discussing ideas, implementations, and future directions. Her insights and mentorship not only helped me advance this project, but also shaped me into a better researcher and inspired me to pursue this path further.

Additionally, I would like to thank Chris Neely, whose comments and insights throughout this project were a significant help in progressing the work, and also helped me deepen my knowledge of hardware design and refine my ideas. I would also like to thank Nachiket Kapre, who provided valuable feedback, learning materials, and resources that made this project possible.

Last but not least, I would like to thank all the members of our lab, whose support and friendship made this journey more enjoyable. I am especially grateful for their helpful comments about my work and for the monthly board game sessions that brightened the busiest times.

Dedication

To my father, whose kindness and support sustained me at every step of this journey.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
List of Figures	ix
List of Tables	x
List of Abbreviations	xi
1 Introduction	1
1.1 Transport Layer Hardware Offloads: Motivation and History	3
1.2 The Evolution of The Transport Protocols	5
1.3 Hardware Programming Framework for The Transport Layer: Why and How?	8
1.4 Existing Hardware Programming Frameworks: Overview and Gaps	9
1.4.1 NanoTransport	9
1.4.2 F4T	11
1.4.3 Tonic	13
1.5 HtraP: A General Purpose Transport Layer Framework	15

2	Transport Processing Shell	17
2.1	Overview	18
2.2	Handling Incoming Requests	19
2.2.1	Events	20
2.2.2	Network and Application Parser	21
2.2.3	Scheduler	22
2.3	Handling Instructions	29
2.3.1	Packet Generator	31
2.3.2	Reassembly	37
3	Transport Processing Engine	41
3.1	Programming in TPE	41
3.1.1	Event Processor Interface	42
3.1.2	Event Processor Implementation	46
3.2	Interface With Transport Processing Shell (TPS)	47
3.2.1	Configuring the Parameters	48
4	Evaluation	49
4.1	Hardware Transport Programming (HtraP) Hardware Feasibility	49
4.2	Micro-benchmarks for the TPS Modules	50
4.2.1	Scheduler	50
4.2.2	Packet Generator	54
4.2.3	Reassembly Module	55
4.3	TPE Programmability	56
4.3.1	Expressiveness	56
4.4	Case Study: A TCP implementation in HtraP	59
4.4.1	Transport Processing Engine (TPE) Implementation	59
4.4.2	TPS Implementation	61
4.4.3	Results	63

5	Conclusion	64
5.1	Summary of Contributions	64
5.2	Future Work	65
5.2.1	Supporting Significantly Larger Number of Flows or Per-Flow Context	65
5.2.2	Enabling Back-Pressure for The Packet Generator	65
5.2.3	Higher level programming framework	66
	References	67
	APPENDICES	72

List of Figures

1.1	Evolution of Ethernet standard speeds by the release date of the corresponding IEEE standard documents.	3
2.1	The overall structure of HtraP	18
2.2	The path each event takes after arriving at the scheduler	24
2.3	The hardware design for the scheduler.	26
2.4	The scheduler pipeline.	26
2.5	The hardware design for the event FIFO memory.	29
2.6	The packet generator algorithm	32
2.7	The packet generator design. The green box is programmable	34
2.8	The design of the reassembly module. The red and blue arrow represent the path for “flush and notify” and “add data segment” instructions respectively	39
2.9	Adding a 256 byte data segment to Reassembly Data Buffer (RDB)	40
4.1	The throughput over time for 1024 flows	52
4.2	Distribution of event scheduling latency for various N-burst traffic patterns for events from 1024 flows over 50 microseconds and a TPE pipeline length of 3.	53
4.3	Packet generation rates	55
4.4	Reassembly module’s instruction processing rate for different packet sizes .	56
4.5	The congestion window size over time	62

List of Tables

2.1	The list of design parameters TPS	20
2.2	A list of the instructions in HtraP	30
3.1	Parameters configured to match TPE	48
4.1	HtraP’s resource utilization for synthesis using the parameters in Table 4.2.	50
4.2	The list of the synthesis values for TPS parameters.	51
4.3	Protocol implementations in TPE	57
4.4	The synthesis result for Transmission Control Protocol (TCP) implementa- tion in TPE	60

List of Abbreviations

AIMD Additive Increase/Multiplicative Decrease [63](#)

BRAM Block Random Access Memory [19](#), [28](#), [49](#), [65](#)

DCQCN Data Center Quantized Congestion Notification [14](#), [58](#), [59](#)

ECN Explicit Congestion Notification [6](#), [7](#), [58](#), [59](#)

FIFO First In First Out [26](#), [28](#), [33–35](#)

FPGA Field-Programmable Gate Arrays [41](#), [49](#), [50](#)

HDL Hardware Description Language [1](#), [8](#)

HLS High-Level Synthesis [8](#), [18](#), [41](#), [42](#), [46](#), [47](#), [56](#), [64–66](#)

HtraP Hardware Transport Programming [vii](#), [ix](#), [x](#), [2](#), [14–16](#), [18](#), [19](#), [21](#), [25](#), [29](#), [30](#), [41](#), [47](#), [49](#), [50](#), [54](#), [59](#), [62–66](#)

iWARP Internet Wide Area RDMA Protocol [4](#), [5](#)

LUT Look-Up Table [49](#)

NIC Network Interface Card [1](#), [4](#), [5](#), [8](#), [20](#), [50](#), [58](#)

QUIC Quick UDP Internet Connections [6](#), [7](#)

RDB Reassembly Data Buffer [ix](#), [37–40](#)

RDMA Remote Direct Memory Access [2](#), [4–6](#), [12](#), [14](#), [58](#)

RMT Re-Configurable Match-Action Tables [22](#)

RoCE RDMA over Converged Ethernet [4](#), [5](#)

RoCEv2 RDMA over Converged Ethernet version 2 [5–7](#), [12](#), [14](#), [44](#), [47](#), [56–59](#), [65](#)

RPC Remote Procedure Call [10](#)

RTL Register-Transfer Level [8](#), [41](#)

RTT Round-Trip Time [6](#)

TCP Transmission Control Protocol [x](#), [4–7](#), [11](#), [12](#), [16](#), [49](#), [50](#), [56](#), [57](#), [59–61](#), [63](#), [65](#)

TPE Transport Processing Engine [vii](#), [x](#), [2](#), [3](#), [15–29](#), [33](#), [36–38](#), [41](#), [46–54](#), [56–60](#), [62](#), [64–66](#)

TPS Transport Processing Shell [vii](#), [x](#), [16](#), [17](#), [19](#), [20](#), [22](#), [29](#), [43](#), [47–51](#), [57](#), [59](#), [61](#)

WAN Wide Area Network [6](#)

WQE Work Queue Element [58](#), [59](#)

Chapter 1

Introduction

As network speeds grow to 100s of Gbps, the traditional software-based processing of the transport layer has become a performance bottleneck. To fully utilize the capabilities of modern high-speed networks, there is a growing interest in offloading transport-layer functionality to hardware, particularly through [Network Interface Card \(NIC\)](#) accelerators [18, 7, 8, 36, 34, 38, 24, 13]. At the same time, the transport layer itself is undergoing significant evolution [5, 39, 16, 20, 40, 23, 33, 10, 14, 17, 15], especially in data center environments where a wide range of transport protocols have been proposed to meet diverse application requirements. These protocols vary in their design goals, from low-latency communication to high throughput or fairness, highlighting the need for flexible and high-performance transport solutions in hardware.

Hardware design is usually carried out using [Hardware Description Language \(HDL\)](#)s such as Verilog and VHDL. However, designing hardware is much more difficult than writing software. The programmer has to think carefully about how to pipeline and parallelize the logic while staying within the hardware resource limits, and debugging such applications often requires working with low-level simulation interfaces. With transport protocols constantly evolving, this is not a one-time effort, as every new protocol either requires starting from scratch or dealing with large and complicated codebases to make changes. This challenge highlights the need for a dedicated hardware programming framework for the transport layer. Fortunately, many transport protocols, despite differing in many aspects, share common functional components such as scheduling, packet generation, and data re-assembly, and typically need to use these components to respond to various events in the transport layer. These components often have similar hardware implementation requirements, which are typically low-level concerns that should not burden protocol designers. The recurrence of such patterns suggests that it is feasible to create a reusable hardware

framework to simplify and accelerate the development of transport-layer functionality in hardware.

There have been several efforts to introduce hardware frameworks for different layers of the network stack. Naturally, similar attempts have been made for the transport layer as well [7, 9, 8]. Some of these efforts focus on a specific class of protocols. For instance, nanoTransport[8] is optimized for message-based transport protocols, such as the ones commonly used in [Remote Direct Memory Access \(RDMA\)](#) environments. Other works, like F4T[9], target variations of TCP and aim to facilitate their implementation through a programmable hardware pipeline. These specialized frameworks take advantage of the properties unique to their target protocols, which allows them to achieve efficiency but also prevents them from supporting a wider range of protocols. In contrast, Tonic[7] attempts to provide a more general-purpose hardware framework for transport programming by identifying common events of the transport layer and providing a programmable hardware framework tailored to respond to these events. However, Tonic’s flexibility is limited because it requires its programmable components to finish in a single clock cycle, making it difficult to implement more complex protocols. This highlights the need for a more general-purpose transport-layer hardware framework that provides greater implementation flexibility and relaxed timing constraints.

Taking inspiration from previous hardware transport layer programming frameworks, we want to explore the possibility of designing a framework that would be capable of handling a wider variety of transport layer protocols. This thesis introduces [Hardware Transport Programming \(HtraP\)](#), a programmable hardware framework for the transport layer. In [HtraP](#), the information about different events in the transport layer is gathered and represented in the form of structs. The programmable part of [HtraP](#), referred to as [Transport Processing Engine \(TPE\)](#), will respond to these events by updating the memory that is responsible for tracking the state of the transport layer, which we will refer to as the context memory. Additionally, it can instruct [HtraP](#) to carry out different actions such as packet generation or data reassembly. In order to allow for a wider variety of transport protocols, we want to fulfill two criteria: (1) allow the [TPE](#) more flexibility and (2) create a well designed interface between the instruction handlers and the [TPE](#) to enable the programmers to describe different actions that might be needed by different protocols without needing to be concerned about the execution details. To achieve the former, we allow the programmer to describe [TPE](#) as a pipeline of an arbitrary length, therefore allowing the [TPE](#) to handle different protocols with various computational complexity. To enable such pipelines, we must address the issue of memory inconsistency. The transport layer needs to handle the context memories associated with different connections. Memory inconsistency happens when a new event has to access the memory that is currently being

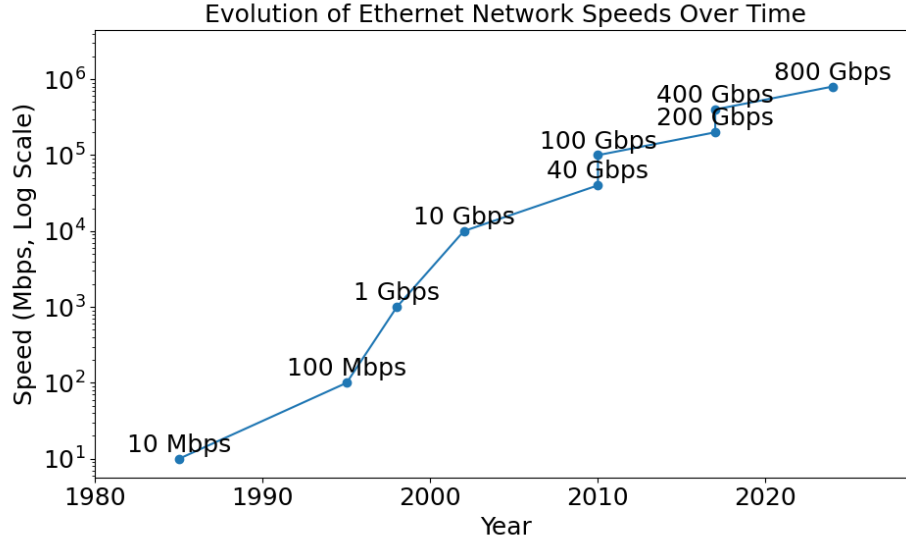


Figure 1.1: Evolution of Ethernet standard speeds by the release date of the corresponding IEEE standard documents.

updated by the [TPE](#). We have designed an scheduler that can sent out events in a way that would avoid such inconsistency. For the second problem, identified the instructions that are commonly used in the transport layer, and designed specific modules to handle them. We made the modules and their instructions general enough to be able to handle a broader set of instructions.

1.1 Transport Layer Hardware Offloads: Motivation and History

Over the past few decades, the network speeds have seen steady and significant increases. Ethernet standards have progressed from 10 Mbps in the early 1980s to 400 Gbps in modern data centers, with the 800 Gbps standard released recently and corresponding technologies currently under development. This upward trend is illustrated in Figure 1.1. To fully utilize these increasing link speeds, the networking devices need to process and generate packets at a higher speed, with packet transfers happening every few nanoseconds.

As networks become faster, software-based approaches to packet processing have strug-

gled to keep up. For example, studies show that in data centers, a significant portions of the CPU cycles are consumed to support the [TCP](#) stack that many applications rely on[19, 28, 37, 9]. It has been observed that a considerable CPU overhead and latency is caused by the operating system kernel’s network stack[19, 28, 37]. Therefore, some works aim to improve the performance of the transport protocols in software through kernel bypasses. For example, some approaches such as mTCP[19] runs the [TCP](#) stack in the user space, while others such as TAS[21], dedicate separate CPU cores to handle certain operations in a faster path. Although these software-based approaches improve CPU overhead and reduce network stack latency, using a hardware-based approach can further improve these overheads and allow programs to take full advantage of higher network speeds.[9]

To address the limitations of software, multiple works have proposed offloading the transport layer operations to hardware.

Offloading protocols originally designed for software. Some works explore offloading parts or all of the [TCP](#) stack to hardware[9, 34, 36, 38, 7]. AccelTCP[34] offloads connection management to the [NIC](#) while keeping the rest of the operations on the host. FlexTOE[38] offloads common [TCP](#) data path operations, such as packet generation and retransmission, to the [NIC](#), while retaining control logic such as congestion control and connection setup on the host. While the aforementioned works focus on offloading parts of the [TCP](#) stack to the [NIC](#), other efforts, such as Limago[36], propose designs that move the entire [TCP](#) stack to a Smart[NIC](#). F4T [9] is a work that introduces a complete [Transmission Control Protocol \(TCP\)](#) offload to hardware and aims to simplify the implementation and modification of different versions of the [TCP](#) protocol. While these efforts specifically target [TCP](#), there are other works that focus on a broader range of protocols. Tonic [7] provides a framework that supports multiple protocols, including software-based ones such as [TCP](#) and [NDP](#) [17]. NanoTransport [8] also targets hardware offloads for various protocols, with a particular focus on protocols for data centers, including [NDP](#) and Homa [33].

Leveraging hardware-based transport protocols. In response to the limitations of [TCP](#), particularly in data centers, solutions based on [Remote Direct Memory Access \(RDMA\)](#) have gained popularity. [RDMA](#) is a network mechanism that enables direct access to the memory of a remote system by having the network stack on the [NIC](#), thereby reducing CPU involvement and enabling lower latency and higher throughput. [RDMA](#) was originally designed for high-performance computing in InfiniBand networks, but it was later adapted for Ethernet-based data centers through the [Internet Wide Area RDMA Protocol \(iWARP\)](#) [3] and [RDMA over Converged Ethernet \(RoCE\)](#) [4] protocols.

[iWARP](#) allows [RDMA](#) operations to run over the standard [TCP/IP](#) stack by encapsu-

lating RDMA requests inside [TCP](#) packets. In contrast, [RoCE](#), which eventually evolved into [RDMA over Converged Ethernet version 2 \(RoCEv2\)](#), assumes a lossless Ethernet network and encapsulates the original InfiniBand transport headers within UDP/IP packets. [RDMA](#) is designed to offload transport-layer operations to the [NIC](#), therefore both of the aforementioned protocols need a hardware implementation. [RoCEv2](#) is often implemented in dedicated hardware, while [iWARP](#) relies on [TCP](#) offloads.

In conclusion, offloading the transport layer to hardware has emerged as an effective response to the growing gap between network speeds and the capabilities of software-based implementations. This approach has managed to improve the capabilities of the networking stack, particularly in data centers and is gaining traction in both industry and research communities, with several recent works exploring hardware acceleration for transport functionalities.

1.2 The Evolution of The Transport Protocols

The transport layer continues to evolve rapidly in response to the different needs of the modern networking environments and applications, such as the performance demands of modern data center networks. Numerous transport protocols have been proposed in recent years, each aiming to optimize specific network characteristics such as latency[5, 6, 40], throughput[16] or fairness[16]. These protocols are designed around different priorities and assumptions, and as a result, they require different functionalities from the underlying system that executes them. For example, some protocols may rely on precise round-trip time measurements, while others demand fine-grained rate control or congestion window mechanisms. This section provides an overview of several transport protocols to highlight the ever-changing state of the transport layer.

The [Transmission Control Protocol \(TCP\)](#) is perhaps one of the oldest protocols still in use today. Despite being designed in 1974 [11], This protocol remains one of the most widely used protocols for reliable data transmission. In this protocol, the sender transfers a stream of data through the network by dividing it into segments, labeling them with a sequence number, and transferring each segment in a packet. It will keep a window to track the state of the sent segments, and updates this window when the receiver confirms the receipt of a segment through an acknowledgment packet, tracking any potential packet loss through duplicate acknowledgments and timeouts. Meanwhile, the [TCP](#) receiver will reassemble the data based on the sequence number and generate an acknowledgment.

Although the core concept of [TCP](#) has mostly remained the same over the years, There have been various attempts to improve different aspects of the protocol and adapt to the

evolving needs of the transport layer. As its limitations became more apparent, many works began to either tailor [TCP](#) for a more specific use, or develop entirely new protocols optimized for specific needs. Data center applications are one of the most common targets of these specializations, as the environment often enables the deployment and testing of new protocols to be more efficient compared to [Wide Area Network \(WAN\)](#). Additionally, data centers have different workload, topology and [Round-Trip Time \(RTT\)](#) compared to [WAN](#), which serves as a motivation for designing new protocols for them. Moreover, data centers can benefit more from specialized transport protocols because their requirements are often more strict and differ from those of general [WAN](#) networks.

TCP Specializations. One of the differences between the data centers and [WAN](#) is the need for latency optimizations. In [WAN](#), the [RTT](#) of the network is often in the order of milliseconds, therefore, unlike data centers, the smaller latency inefficiencies in the [TCP](#) implementation will not cause a notable problem for them. This means the [TCP](#) improvements can make a significant difference in the data centers. For example, [TCP](#) detects congestion primarily through packet loss caused by buffer overflows, which can result in significant queuing delays. While this may be less noticeable in [WAN](#), where high propagation delays already lead to large [RTTs](#), the effect is much more pronounced in data centers. Because in data centers [RTTs](#) are low and queuing delays become a major concern. To address this, several protocols[5, 6, 40, 35] have been developed that use alternative signals, such as [Explicit Congestion Notification \(ECN\)](#), to detect and mitigate congestion more effectively.

Protocols based on Remote Direct Memory Access (RDMA). As mentioned in Section 1.1, [RoCEv2](#) is an [RDMA](#) based protocol commonly used in data center networks. However, [RoCEv2](#) is not designed to handle congestion and its performance would see significant drops in these situations. To address this, multiple bodies of work have focused on developing a congestion control algorithm for [RDMA](#) systems. Some works perform the task by adjusting the data transfer rate at the sender[44, 29, 25, 22]. [DCQCN](#)[44] is one example of such protocols, which adjusts the sending rate with the help of the congestion notification packets generated periodically at the receiver. Other approaches such as [MP-RDMA](#)[26] seek to improve congestion by load balancing.

Quick UDP Internet Connections (QUIC). Data centers are not the only domains that can benefit from specialization, [QUIC](#)[23] is a protocol designed to address [TCP](#)'s shortcomings in providing latency-sensitive network services such as video streaming, by identifying the underlying problems that make latency optimization difficult for [TCP](#). In addition to incurring latency through the handshaking process, [TCP](#) connections often use a single stream of bytes for communication, which limits the ability of latency-sensitive applications to differentiate between types of traffic. This can lead to [TCP](#) incurring

unwanted delays due to head-of-line blocking, which occurs because all independent messages for a single connection have to share the same queue and in case of a packet loss for one of them, the delivery of every message will be stalled due to the retransmission process. [QUIC](#) was implemented to address the limitations of [TCP](#). It introduces stream abstractions for handling multiple requests within a single connection and places frames from different streams into individual packets. This design prevents different independent requests within the same flow from blocking each other. Additionally, [QUIC](#) offers faster connection establishment and enhanced security features.

As networks evolve and gain new capabilities, they create new paths for designing transport protocols that can leverage these advancements. A notable example is the emergence of programmable network switches, which have introduced new opportunities for transport layer optimization. Traditionally, some transport protocols relied on limited support from network devices, such as the [ECN](#) bits for congestion control[5, 6, 40, 35]. In contrast, programmable switches enable far more flexible operations, allowing transport protocols to be co-designed with the switch logic itself[17, 25]. As before, data centers are the primary target of these optimizations, as they can support the deployment of programmable switches and have full control over their configuration.

[NDP](#)[17] is one example of such protocols. In NDP, switches are programmed to trim packets instead of dropping them when the link becomes congested. A trimmed packet contains only the header, which is then forwarded to the receiver as usual. Upon receiving such a header, the receiver generates a negative acknowledgment to inform the sender of the packet loss. Using these trimmed headers, the receiver can infer the state of the network and act accordingly. Additionally, the sender no longer needs to rely on transmission timeouts to detect congestion.

Looking at this trend, we can conclude that the transport protocols and the transport layer will keep evolving. As network capabilities are increased, the protocols will change to take advantage of them and adapt to the corresponding challenges, as seen with the example of [TCP](#)s congestion control evolution, or the NDP protocol leveraging the programmable switches in the data centers. Additionally, as new applications with different performance and functionality needs emerge, transport layer will need to optimize for these specific needs, as seen through the examples of [QUIC](#) for latency-sensitive applications and [RoCEv2](#) for the data centers.

1.3 Hardware Programming Framework for The Transport Layer: Why and How?

As seen in section 1.1, there has been a growing interest in offloading the transport layer operation to hardware, as this can reduce the CPU overhead and improve the latency. However, developing programs for hardware is a more challenging task than software. In order to have an efficient hardware implementation, the design needs to be carefully pipelined to execute at line rate and meet the strict timing requirements of network applications. Additionally, programming in hardware often involves working with low-level interfaces and testbenches, which adds difficulty to debugging and maintaining a hardware codebase. Due to these challenges, developing a hardware solution for network has been an unpopular choice, despite its benefits[13].

Additionally, as seen in Section 1.2, Transport layer protocols are constantly improving, and new protocols are being introduced, which means that designing these hardware programs won't be a one-time effort, and the code would need to be constantly updated and maintained. For example, it has been observed that TCP Offload Engines (TOE)s failed to gain widespread adoption due to their inflexible nature. Therefore, any program developed for the transport layer is likely to need to undergo some changes as the time passes, and updating an HDL based code is much more difficult than software based ones.

One of the solutions to the difficulty of programming in HDLs are High-Level Synthesis (HLS) tools. With these tools, the users are able to describe their desired hardware in a high-level language such as C++, and the HLS tool will later translate it into a Register-Transfer Level (RTL) implementation. These tools often facilitate the process of testing and development for the hardware developers. However, fully developing an efficient system in HLS is not much easier, as it often requires extensive knowledge of the HLS optimization commands and often provides less flexibility in controlling resource usage and latency, which makes it more difficult to fit the design into strict line-rate and limited resource NICs required by the transport protocols.

However, looking at different examples of transport layer protocols, it has been observed that despite their differences, they often share lots of similarities.[7, 8] These similarities show that, with correct abstraction, it would be possible to simplify the programming of the transport layer in hardware, and allow the programmers to mostly focus on implementing the control logic of the protocol instead of concerning themselves with low-level implementation and optimization of common data path modules.

In conclusion, as network speeds continue to increase, offloading transport functionalities to hardware has received growing attention. However, the complexity and rigidity of

current hardware design flows pose significant barriers to experimentation in this space. This, along with the evolving nature of the transport layer, highlights the need for a dedicated hardware programming framework that simplifies the implementation and modification of transport-layer protocols. Such a framework would allow researchers and developers to rapidly explore new ideas while retaining the performance benefits of hardware-based transport.

1.4 Existing Hardware Programming Frameworks: Overview and Gaps

Ever since the introduction of SmartNICs, new opportunities have risen for offloading network functionalities to hardware. We have discussed some of the offloads in Section 1.1. As discussed in Section 1.3, there has been a growing interest in designing hardware programming frameworks for network functionalities. In this section, we will look at some of these frameworks, their goals, and their limitations. we examine three examples of prior work. Since there are not many works that purpose a general purpose framework for the transport layer, we will first discuss two examples of specialized programmable designs: nanoTransport[8] and F4T[9]. We explore their capabilities and explain why adapting them into a general-purpose framework is non-trivial and may hurt their performance. Finally, we examine Tonic, its programming model, and some of its limitations in supporting a broader range of protocols and supporting faster clock rates.

1.4.1 NanoTransport

As mentioned in Section 1.2, data centers are one of the environments that can benefit greatly from performance improvements due to their stringent throughput and latency requirements. Therefore, some bodies of work focus on improving the performance for the data center environment. NanoTransport[8] is one of these works. It is a programmable transport layer architecture that aims to simplify the process of developing message-based transport protocols in hardware. This architecture uses P4 pipelines to process incoming and outgoing packets. To implement the protocol and decide the proper response to each packet, these pipelines need to have information about the current state of the transport layer. To track this state, each stage of the pipeline has its own small memory that it can use for making decisions and update after each stage. For incoming packets, after updating the memory, the pipelines may create a *Control Packet Event*. This event will

be sent to a packet generator module we will mention later to trigger the generation of a control packet such as acknowledgment. After this stage, The data packet is then passed to the reassembly module, which stores it and checks whether the full message has been received. The reassembly module also maintains a per-message timer; if the timer expires, it indicates a reception failure and the corresponding state is discarded. If the reassembly module detects that all packets of a message are received, they are reassembled in order and forwarded as a complete message to the CPU or RDMA engine. For outgoing packets, The messages that are received from the application will go to a packetization module that is responsible for dividing them up into segments. This module will then transfer these segments and set a retransmission timer for each. The timer will trigger the retransmission of these segments upon timeout. Additionally, the packet generator module mentioned in the previous paragraph may also generate a control packet. The P4 pipeline will receive the packet from either the packetization module or the packet generator, add the proper headers, and send the packets out.

By using the existing P4 abstractions for the events and relevant processes, NanoTransport takes advantage of the existing P4 programming framework. While NanoTransport works well for handling [Remote Procedure Call \(RPC\)](#)s, which was the main goal of the work, we argue that it would be difficult to adapt this model to handle a wider variety of transport protocols.

Stateful memory. NanoTransport uses P4 to handle the packet processing stage of the transport layer, and any stateful operation is also done through careful use of memory update primitives. These primitives can be used in each stage of the P4 pipeline. However, the size of the memory available to these pipelines is small and different stages of the pipeline cannot share the memory with each other. This can limit the ability of the user to implement more complex algorithms that require a more complicated state memory update.

Timer. NanoTransport has two timers, one in the reassembly module that is supposed to discard the message after timeout, and another one that triggers the retransmission of the sent packets on timeout. The functionality of these timers are fixed, and NanoTransport does not offer any other timer module. This can cause difficulty with algorithms that would need to use timers for a different purpose, such as DCQCN[44], which requires a timer to update the state variables that will be used to determine the transmission rate of each flow.

Non-message-based protocols. NanoTransport is designed to accelerate message-based protocols, making it unsuitable for adapting other types of protocols, such as stream-based TCP. In stream-based protocols, each flow has one or more send buffers that the

application data is added to irrespective of message boundaries, and the transport protocol is responsible to deliver the stream in the same order to the other communication end point. That is, the transport layer just sees a stream of bytes, not a sequence of messages with a known size. One potential idea to implement stream-based protocols with NanoTransport is to treat the entire contents of the send buffer as a message; however, this would create problems with the unbounded size of the [TCP](#) streams. To implement [TCP](#), another external module would likely need to track the data currently present in the packetization module, and send the remaining portion of data to the reassembly module when needed. While this approach might enable NanoTransport to support TCP, adding this external support is a non-trivial effort.

1.4.2 F4T

As mentioned in section [1.2](#), there are many different versions of [TCP](#) that aim to optimize the protocol for different purposes. This has led to some works aiming to write a hardware programming framework for [TCP](#)[\[38, 9\]](#). These programming frameworks often take advantage of the shared characteristics between [TCP](#) protocols, such as the byte-stream and sliding window abstractions. F4T[\[9\]](#) is one of these frameworks.

F4T receives metadata related to application requests and network packets in the form of events, which are processed by its Flow Processing Cores (FPCs). The FPCs require access to the current transport-layer state to accurately handle incoming events. In the context of the [TCP](#) protocol, this state includes information such as the window size and the most recently received acknowledgment. The state memory must be updated after each processing step. If updating the state memory takes more than one cycle, a new event might arrive that needs to access the same memory location while it is still being updated. In such situations, memory access must wait until the updated value is available, resulting in a stall.

To improve performance, F4T aims to reduce stalls caused by read-modify-write operations on shared state memory. It maintains a list of 128 active flows and iterates over this list, reading one entry per cycle. This design ensures at least 128 cycles between two accesses to the same memory location, and since the FPCs usually process events in fewer than 128 cycles, conflicts are generally avoided. However, a flow may receive multiple events while waiting for its turn. In these cases, F4T accumulates information from the incoming events. For example, if duplicate ACKs are received, it increments a counter and discards the redundant events. This accumulation must happen in a single cycle to avoid memory inconsistency issues.

While the total number of active flows is typically on the order of 10,000, F4T only maintains 128 flows in memory at a time. To scale, it introduces a mechanism to evict less active flows from the event table and bring in others. This swapping mechanism enables F4T to support a much larger number of flows.

Although this approach enables F4T to implement [TCP](#) efficiently, it is difficult to adapt the architecture to other transport protocols.

Adapting to other protocols. F4T's method of accumulating the event information would be difficult to adapt for a more general purpose framework since the process of accumulating the information may require more than one clock cycle. If the process takes more than one clock cycle, the memory inconsistency problem that F4T was trying to avoid would return and introduce complexities to the design. Next, we will look at the two categories of protocols that would be difficult to adapt to this method.

- **Message based protocols.** The event accumulation will cause problem with message-based protocols such as [RoCEv2](#). We will look at one example with the send operations. In [RoCEv2](#), messages are designed to be processed independently. However, they all arrive via the same link and belong to the same flow. Therefore, it would be difficult to accumulate the information of these events into one event. Additionally, Each [RDMA](#) send must match a corresponding posted Receive on the receiver side, which means that accumulating the events inside one would complicate this matter as well.
- **Time based congestion control algorithms.** Time-based congestion control algorithms are a class of transport-layer mechanisms that adjust sending rates based on timing information rather than relying primarily on packet loss or explicit feedback. These algorithms infer network congestion levels by measuring how long packets take to traverse the network, especially focusing on delay trends over time.

Some time-based congestion control algorithms rely on complex estimation procedures that update internal state variables using the most recent round-trip time measurements. These updates sometimes involve arithmetic operations such as e^x or other complex functions[41, 43, 12], which would take more than one clock cycle to compute. As a result, it would not be possible to accumulate the data of these RTT values within one event in one clock cycle.

1.4.3 Tonic

Tonic is a hardware programming framework for the transport layer that aims to support a wide variety of transport layer protocols. The authors of Tonic observe that, despite differences in sender logic across various transport protocols, many recurring patterns and common operations exist, such as the need to provide congestion control mechanisms, generate packets and handle their status, process different types of acknowledgments, and manage timers. These observations led to the proposal of a hardware framework for the transport layer.

Similar to the previous modules, Tonic also receives the metadata about various events of the transport layer and responds to them accordingly. Since Tonic focuses on the sender side, its main responsibility is generating data packets, and all events are used to guide this process. Tonic is divided into two main parts: the *data delivery engine* and the *credit engine*. The former is mainly responsible for deciding *what* to transfer, which is done by generating the address of the segments that should be transmitted. The latter is responsible for deciding *when* to transfer, which is done through the use of per-flow credits.

To generate segment addresses, the **data delivery engine** relies on a series of bitmaps responsible for tracking the status of the transmission window. Each element in a bitmap corresponds to one segment and indicates the status of that segment, such as whether it is currently in flight or needs retransmission. Therefore, to correctly generate segment addresses, the engine must maintain and update these bitmaps according to the incoming events and the current state of the transport layer, which is stored in a per-flow state memory. The authors identify four different events that can cause updates to these bitmaps: (1) *timeout*, which often indicates a packet loss; (2) *control packets* such as acknowledgments; (3) *segment generation*, which changes the status of segments after they are generated; and (4) *segment transmission*, which marks a segment as *in-flight*. There are four corresponding modules responsible for handling each of these events. Because all of these modules update the state memory, they must complete their operations in a single cycle to avoid the memory inconsistency issue discussed in previous subsections. The modules handling (1) and (2) are programmable and shape the policy of a protocol, while (3) and (4) are fixed.

The **credit engine** is responsible for deciding when to transfer the segments. To make this decision, it maintains per-flow *credits*—a number indicating how many bytes a particular flow can transmit at any given time. There are different ways to update the credit, and the updating scheme can be used to implement different scheduling policies. Tonic allows programmers to choose between three credit update schemes: (1) *sliding window*, (2) *rate-based*, and (3) *grant-based*. The credit engine will transmit a segment

once sufficient credit is available.

HtraP builds on the same motivation but takes the idea of transport programmability further. Rather than confining users to modify logic within a rigid pipeline, **HtraP** allows users to define their own processing logic across a multistage pipeline. This design enables a broader variety of transport algorithms to be implemented in hardware, including those with more complex scheduling, flow management, or data handling behavior. By expanding the scope of programmability, **HtraP** aims to support a richer set of protocols and to offer finer-grained control over packet processing and flow coordination than what is possible in Tonic. We will briefly explain the ways that **HtraP** improves upon Tonic.

Focusing only on the sender side. Tonic is more focused on implementing the sender side and does not provide the means to implement programs commonly required on the receiver side, such as acknowledgment generation and data reassembly. By offering a more flexible event processing unit and a dedicated reassembly module, **HtraP** allows the programmer to implement both sender and receiver logic within the same hardware.

Lack of flexibility in segment generation. While Tonic provides programmable modules for handling acknowledgments and timers, it uses fixed logic for other components. We believe this fixed logic limits the number of transport protocols it can fully implement. Consider **RoCEv2** as an example: Tonic’s programmable modules allow users to faithfully implement the **Data Center Quantized Congestion Notification (DCQCN)** algorithm used by **RoCEv2**. However, its fixed segments are not suited for handling **RDMA** logic. As a message-based protocol, **RoCEv2** issues independent memory access requests through discrete messages, which means that the memory location accessed by these messages would be independent of each other. In contrast, Tonic’s segment generation module assumes it is selecting segments from a continuous buffer of data and calculates the address of the next segment accordingly. This assumption is incompatible with the independent nature of messages, as two requests from the same flow may need to access different memory regions.

To handle this scenario, the segment generation logic would need to recognize message boundaries and the unique metadata associated with each message. Given the strict one-cycle constraint imposed on this module, implementing such changes would be non-trivial and may not be feasible within Tonic’s current architecture. This suggests the need for a more flexible structure to support such operations.

Handling faster clock rates. Because Tonic needs its programs to execute in one cycle, it would be difficult to adapt it to faster clock rates, increasing the clock rate would impose an even stricter constraint on the programs. One possible way for tonic to support more programs in faster rates is to allow the programs to be pipelined. However, in that case it would also need to address the issue of data hazards.

1.5 HtraP: A General Purpose Transport Layer Framework

This thesis presents HtraP, a general hardware programming framework for the transport layer. Similar to the prior work, [HtraP](#) uses events as an abstraction for the events in the transport layer. The events can come from three sources: The application, the network and the timer. There is no limitation to the definition of events and the user can define parsers to create these events from external inputs to the system, such as packets and application requests.

The users will program an event processing pipeline, called [TPE](#) that can process these events, issue instructions and update the state variables. The pipeline can be of any length. We improved upon Tonic by allowing the programmable part to take more than one cycle. To address the data hazard issue that prevented Tonic from using longer pipeline, we implemented an event scheduler that tracks the list of “safe flows” (i.e. flows that have no event in the pipelines) and only chooses the next event from them, therefore avoiding any memory inconsistency.

Additionally, the [TPE](#) can freely modify the state memory, and there is no fixed logic that it needs to adhere to, as long as the [TPE](#) program can be pipelined. This way, we can support a broader range of programs, and improve upon nanoTransport, who had a more limited access to the shared memory due it being scattered across pipeline stages.

The [TPE](#) can also communicate with external modules to generate packets, reassemble the data or set timers. These three modules are mostly fixed save for a small programmable section in the packet generator. However, the [TPE](#) programmer has the freedom to decide how or when to issue commands to these modules. In this way, we can address some of the flexibility issues mentioned in Section 1.4 and allow users to implement a broader range of programs. However, the interface of these designs plays an important role in allowing such flexibility.

Specifically, the **packet generator** accepts two types of instructions: Segment generation and scheduling update. The former can trigger the generation of one or several packets while the latter updates packet scheduling parameters such as rates. The **data reassembly** module has three types of instructions: Allocating data buffers, writing data from incoming packets into the a specific location in the buffer, and sending the reassembled data to the application. This module allocates the buffers to flows upon request, and has to maintain and update the said buffer after receiving instructions. Finally, the **timer** module is responsible for tracking time and the [TPE](#) can ask it to start, restart or cancel a timer.

Thesis outline. In the following chapters, we will describe the design details of [HtraP](#), and implement a prototype to evaluate the feasibility and effectiveness of this design. Specifically, we will explain the proposed hardware design, which we will refer to as [Transport Processing Shell \(TPS\)](#), which provides [TPE](#) with events and receives its instructions in Chapter [2](#). Chapter [3](#) will focus on the programmable part of the design and its interfaces. For the evaluation in Chapter [4](#) We will implement a prototype of our design and evaluate the performance of [HtraP](#) and each individual module. Additionally, we will implement three protocols in [TPE](#) to test the effectiveness of our interface. Finally, we will fully configure [HtraP](#) for the [TCP](#) implementation and test the correctness of the implementation. Finally, we will discuss potential future directions for this work in Chapter [5](#).

Chapter 2

Transport Processing Shell

The transport layer is responsible for a wide range of tasks, including packet parsing and generation, flow control, handling incoming data, managing retransmissions, and interacting with the application layer. These tasks involve both operations such as formatting packets and maintaining a per-flow state, which needs to move and handle data, and decision-making operations, such as when to transmit, retransmit, or drop packets based on protocol-specific logic and the current state. As such, any hardware offload targeting the transport layer must be able to perform these functions efficiently and at scale, without introducing bottlenecks or compromising correctness.

To enable flexible and programmable transport-layer offload, we propose a separation between the processing engine ([TPE](#)), which is responsible for policy and decision making, and the execution body, which is responsible for performing the actual tasks such as packet generation and reassembly. The execution body, which we will refer to as [Transport Processing Shell \(TPS\)](#), exposes a structured interface that allows [TPE](#) to receive a report that contains the current connection state and the specifics of an event or request. The [TPE](#) then determines the appropriate course of action, such as setting a timer, transmitting a segment, or modifying the internal state, and issues the corresponding command. [TPS](#) performs the requested actions and updates the per-flow state memory accordingly. This separation enables efficient hardware acceleration without sacrificing programmability, allowing developers to modify transport behavior at the policy level without requiring changes to the logic of the common operations.

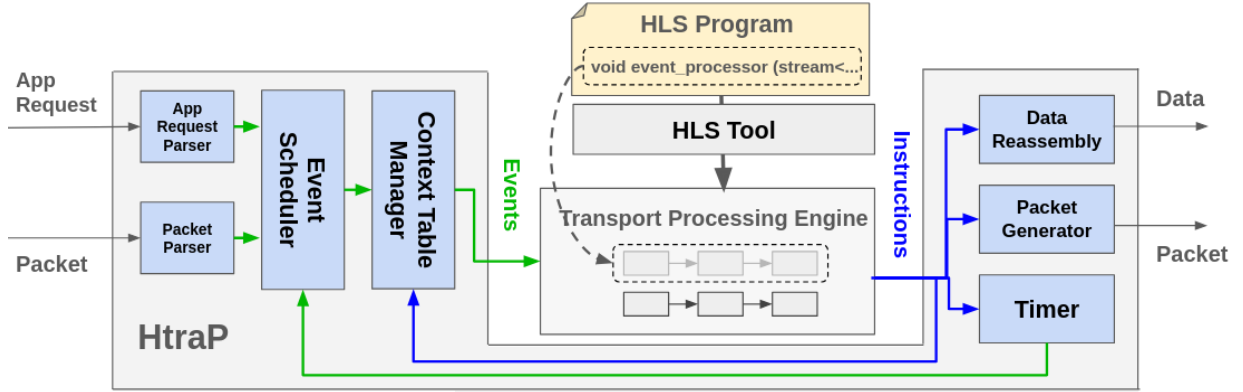


Figure 2.1: The overall structure of [HtraP](#)

2.1 Overview

In [HtraP](#), most of the programming happens in [Transport Processing Engine \(TPE\)](#), but some parts of the other modules are also minimally programmable to allow better adjustments to different protocols. An overview of the hardware architecture can be seen in [Figure 2.1](#).

[HtraP](#) can receive inputs either from the network, in the form of packets, or from the application interface. Network packets are handled by the network parser, while application requests are processed by the application parser. Each parser identifies the flow to which an incoming event belongs, extracts and condenses the relevant information needed by the [TPE](#), and forwards this condensed representation to the next stage. We refer to this condensed representation as an “Event.” [Section 2.2.1](#) provides a more detailed discussion of these Events. The events created by the parsers will arrive at the event scheduler. The scheduler will store these events and choose the next event to be processed by [TPE](#). Once an event is selected, it is sent to the context table manager. Each row in the context table represents the state of a single flow. Using the flow ID included in the event, the context manager fetches the corresponding row from context memory. The event, along with its associated context, is then passed to the [TPE](#).

The [TPE](#) is a component defined by the programmer using [High-Level Synthesis \(HLS\)](#). [HLS](#) allows the programmers to define hardware logic in a high-level language such as C++ and synthesize it for hardware. The [TPE](#) will be described in detail in [Chapter 3](#). Upon receiving an event, the [TPE](#) calculates the new values for the context, which will replace the

old ones after the process is finished. In addition, it may generate instructions for other modules. These instructions are then sent to the relevant modules, which include the Packet Generator, Data Reassembly, and Timer modules. The Data Reassembly module will reassemble the data inside buffers and send them to the application upon request. The Packet Generator is responsible for constructing and scheduling outgoing packets, while the Timer module initiates timers that will trigger new events upon expiration.

Per-Flow Context Memory. The context memory is responsible for keeping the relevant state data for each flow. Before entering [TPE](#), the events will access the context memory to get the state information, and will also access it after the process to update it with a new value. The context memory is implemented using a [Block Random Access Memory \(BRAM\)](#).

TPS Configurable Parameters. When designing [HtraP](#), we parameterized most of the architecture to ensure that it can be adapted to different scenarios and system requirements. The values of these parameters can significantly affect the system’s memory usage. Therefore, users should be able to adjust various design parameters to suit their specific use cases. For example, the total number of flows can be configured based on the expected traffic patterns.

Additionally, certain parameters such as the event width or context width, are determined by the user’s implementation of the [TPE](#). A future version of [HtraP](#) would automatically set these parameters based on the design of the [TPS](#).

A complete list of parameters in the design of the [TPS](#) is reported in [Table 2.1](#).

2.2 Handling Incoming Requests

[HtraP](#) begins by receiving packets or application-level requests as input. These inputs must ultimately be processed by the [TPE](#), which serves as the processing unit responsible for issuing transport-layer commands and calculating the new values that [TPS](#) will use to update the context memory. However, before reaching the [TPE](#), each request undergoes a structured processing pipeline that transforms, classifies, and schedules it for subsequent processing.

This section outlines the path taken by each request to reach [TPE](#). In [Section 2.2.1](#), we introduce events, which serve as the internal representation of incoming requests throughout the pipeline. [Section 2.2.2](#) then describes the role of the parsers, which analyze the raw input and generate corresponding events. Finally, [Section 2.2.3](#) discusses the scheduler,

Module	Parameter	Set By	Description
All of the Design	FLOW_COUNT	User	Total Number of flows
	EVENT_WIDTH	TPE	size of the event in bits
	EVENT_TYPE_COUNT	TPE	total number of event types
	CONTEXT_WIDTH	TPE	size of the context in bits
	SERIALIZED_DATA_WIDTH	NIC	The width of the memory transfer link
	SERIALIZED_PACKET_WIDTH	NIC	The width of the network transfer link
Scheduler	PER_FLOW_MEMORY_DEPTH	User	the size of the per flow queues in event memory
	EVENT_QUEUE_DEPTH	User	the size of the safe queues
Packet Generator	HEADER_WIDTH	TPE	size of the header in bits
	INSTRUCTION_QUEUE_DEPTH	User	the size of the instruction queues
Reassembly module	PER_BUFFER_MEMORY_DEPTH	User	the size of the buffers allocated to each flow

Table 2.1: The list of design parameters [TPS](#)

which manages the ordering of events and selects the next event to be dispatched to the [TPE](#) for processing. Through this multi-stage pipeline, the system prepares the incoming requests to be handled by the [TPE](#).

2.2.1 Events

Events serve as structured abstractions of transport-layer activity, capturing information derived from both external inputs, such as network packets or application-generated requests, and internal ones, such as timeout. Each event corresponds to a distinct event within the transport-layer program and provides an interface for conveying relevant data through the processing pipeline.

HtraP can be programmed to transform inputs into user-defined events. Protocol designers can provide **HtraP** with the set of events defined in a C struct-like format, allowing them to extend the structure with fields relevant to their specific use case. The users can define different events corresponding to different types of transport layer events. For example, the TCP protocol may have two types of events for the network inputs: acknowledgment and data. An example of an event definition can be seen in Listing 2.1.

```
struct ACK: public event_t {
    ap_uint<32> ack_seq;
    ap_uint<32> rwnd_size;
};
```

Listing 2.1: Event processor interface

HtraP's parser will be configured according to the user's program to set these fields properly after receiving an input, and **TPE** will later use the information in these fields to take appropriate action. In addition to the user defined fields, the event that travels through the incoming event pipeline has two extra fields that will help the other modules in carrying out their tasks: type and flow id. The type field categorizes the event and corresponds to one of the user-defined event types. It is used to inform the **TPE** on how to interpret the contents of the event and process it. This enables the **TPE** to dispatch the event to its dedicated processing logic. The flow id field associates the event with a specific flow context, ensuring that state lookups and updates are correctly scoped. Additionally, it helps the scheduler in ensuring fairness across flows and in avoiding data races.

2.2.2 Network and Application Parser

HtraP's parsers are responsible for receiving incoming requests from either the application layer or the network and transforming them into structured events that can be processed by the **TPE**. Each parser extracts relevant fields from the request according to a user-defined specification and constructs a new event containing this information. These events serve as the unified input format for subsequent components, such as the scheduler and **TPE**. Additionally, the parsers will assign a unique flow id to each request based on the value of certain fields, which will help the subsequent modules in fetching the correct state for this flow. The network parser will store the payload of the incoming packet in a memory we will refer to as the *Temporary Data Memory*. The network parser has access to the address where this data is stored and can assign it to one of the fields of the event. This memory will later be accessed by the reassembly module to retrieve the data.

Event Generation

The structure of the parser logic closely resembles the match-action paradigm commonly used in programmable data planes, such as those defined in the [Re-Configurable Match-Action Tables \(RMT\)](#) architecture. Parser programs consist of a set of rules that examine specific header fields and generate corresponding events when certain conditions are met. This design allows developers to specify fine-grained control over how different types of requests are parsed and categorized. To implement this functionality, we leverage existing hardware components inspired by programmable switches, adapting their mechanisms for use within the transport-layer context. This not only simplifies implementation but also benefits from the efficiency and scalability of high-speed programmable packet parsing techniques.

Flow ID Mapping

In addition to generating events, the parsers are responsible for assigning each request to a unique flow context. This is achieved by extracting user-specified fields from the request—such as source and destination addresses, ports, or other identifiers—and mapping them to a flow identifier (flow id). The mapping is implemented using a hash table that associates these extracted keys with internal flow IDs. One way to ensure efficient and deterministic lookup performance, would be to employ a cuckoo hashing scheme, which has been previously proven effective for this task [\[9, 36\]](#).

2.2.3 Scheduler

Event scheduling is a fundamental component of the [TPS](#), tasked with selecting events from multiple sources such as application, network, or timer, and dispatching them to the [TPE](#) for processing. The scheduler operates under tight constraints, as it must maintain high throughput while avoiding any data hazard. The processing of each event involves reading from and writing to a shared context memory that stores per-flow state. To maintain state consistency, the scheduler must ensure that no two events from the same flow are active in the pipeline concurrently.

The processing of each event will require reads and writes to certain subsets of the corresponding flow’s state in the context memory. As such, the TPS needs to read the proper flow context before sending the event into the [TPE](#) pipeline, and write the updated state after [TPE](#) is done processing the event. We want the [TPE](#) pipeline to be full at all

times, and this leads to an important requirement: only one event per flow may be in the pipeline at any given time. A data hazard, or memory collision, occurs if an event attempts to access context state that is currently being modified by another in-flight event from the same flow. We will refer to the events inside the [TPE](#) as active events. A conservative solution to this problem is to stall the pipeline or reject new events until the hazard clears. However, this approach limits throughput and increases tail latency under high load.

To avoid stalling, the scheduler tracks the status of each flow. When selecting the next event to schedule, the scheduler only chooses the flows with no active events, ensuring that only safe, non-conflicting events are dispatched. Given that the number of active flows in realistic networking workloads typically far exceeds the number of pipeline stages in the [TPE](#), it is statistically likely that there will always be at least one eligible flow with no active event that can enter the [TPE](#) without causing any state inconsistency. This design allows the scheduler to maintain high utilization of the pipeline without introducing data hazards.

Logical Diagram

The scheduler has an event memory that stores the events for each flow in a different buffer. When events reach the scheduler, they are inserted into the event memory, and their associated flow ID is used to guide the scheduling process. To prevent memory collisions and ensure correct access to shared context state, the scheduler maintains a list of “safe flows”. This list contains all flows that can access memory without causing data hazards, and each cycle, chooses the next flow from the list in a round-robin fashion. Therefore, the main task of the scheduler is maintaining and updating the list with each new arrival. In order to accurately update the list of safe flows, the scheduler needs to keep track of two pieces of information for each flow: (1) whether or not the flow is currently active in the pipeline and (2) whether the flow has any pending events in memory. It will then use this information to accurately update the list of safe flows.

Figure [2.2](#) illustrates the path an event and its associated flow ID follow within the scheduler, as well as how the scheduler maintains and updates the list of safe flows. When an event arrives, it is first stored in the event memory. This memory structure maintains a per-flow queue of pending events and ensures that events for each flow are processed in the order they arrive. Once the event is stored, its flow ID advances to the next stage of the scheduler.

The scheduler keeps an information table for each flow to track the two previously mentioned pieces of information: (1) whether the flow is currently active in the pipeline

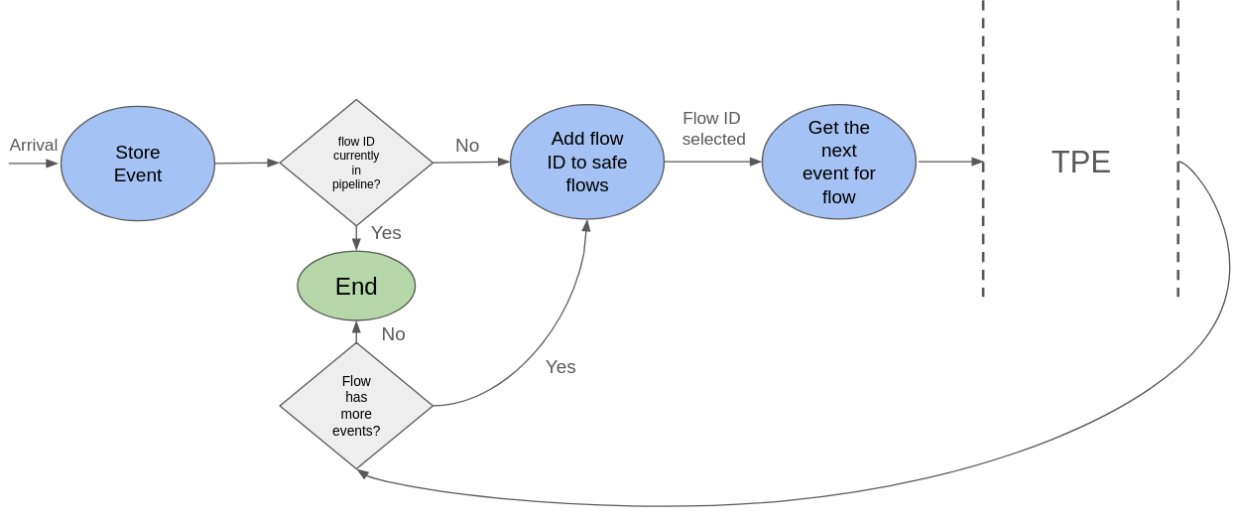


Figure 2.2: The path each event takes after arriving at the scheduler

and (2) whether the flow has any pending events in memory. The scheduler will use this table to determine the state of a flow. The way this table is updated will be explained in the *Hardware Design* section.

At this stage, the scheduler accesses the information table to check whether the corresponding flow is currently active in the processing pipeline. If the flow is not active, it is added to the list of safe flows, making it eligible for scheduling. When the scheduler eventually selects this flow, it accesses the memory to retrieve the next pending event for processing. After the event completes its execution in the TPE, the flow ID is returned to the scheduler, which then checks whether any additional events remain in the flow's queue. This check is performed by accessing the information table and using a flag carried by the event, the details are explained in the *Hardware Design* section. If the flow has remaining events, it is re-added to the list of safe flows, allowing the next event to be processed in turn. This cycle ensures that events are handled in order while avoiding conflicts in memory access.

To establish the correctness of our event scheduling algorithm, we define two key criteria: (1) at any point in time, there is at most one event per flow present in the processing pipeline, and (2) all events are eventually processed. These criteria ensure that there is no memory inconsistency and guarantee that no event will be lost.

To show that only one event per flow can be in the pipeline at a time, we consider how

a flow enters the pipeline. The only way this can happen is through the arrival of a new event. However, before a flow is allowed to proceed, the scheduler checks the information table to see if it is already active in the pipeline. If it is, the flow is held back and will not join the safe flow list for scheduling. This ensures that two events from the same flow cannot be in the pipeline at the same time, preventing simultaneous access to shared flow state.

To show that all events eventually get processed, we rely on tracking the remaining events for each flow. After an event finishes processing, the system checks if there are more events waiting for the same flow. If so, the flow is immediately added back to the safe flows list and scheduled again. This ensures that after a flow enters the pipelines, it will not leave until all of its pending events have been processed. If the flow has no more events, it is removed from the active set. If new events arrive for that flow after it has left the pipeline, it would be treated as inactive and allowed to re-enter the pipeline. This design ensures that every event will eventually be processed.

Hardware Design

To implement the scheduling algorithm, we designed the hardware architecture shown in Figure 2.3. The scheduler operates as a five-stage pipeline shown in Figure 2.4. In the first stage, it accesses and updates the flow’s state in the “Information Table”. In the second stage, the flow is added to the set of safe flows if it satisfies the required conditions. The third stage selects a flow from the safe flows for processing. Finally, the fourth and fifth stages handle the reading of the next event from the “Event FIFO Memory”.

The scheduler module has two inputs: (1) new events and (2) returning flows. New events refer to those generated by the event-generating modules in [HtraP](#), namely the parsers and the timer. Returning flows are the flow ID of the events that the [TPE](#) has processed in the previous stage. The output of the scheduler is the next event to be processed.

Safe Flow Queues. We need to maintain a list of safe queues and select a flow to send to the [TPE](#) each cycle. This functionality is implemented within the safe flow queues module. Each cycle, the scheduler can receive up to two additions to this list. These additions may come from returning flows, new event flow IDs, or both. However, these inputs do not always result in a new event being added. Since the number of possible inputs per cycle can exceed the number of outputs, the scheduler must include a queue to buffer the extra inputs and process them later.

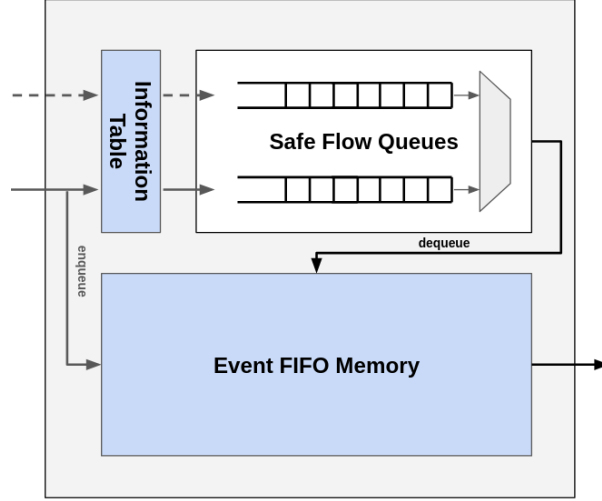


Figure 2.3: The hardware design for the scheduler.



Figure 2.4: The scheduler pipeline.

To ensure fairness across different flows, the scheduler selects the next flow from the safe flow list in a round-robin fashion. This round-robin mechanism is implemented using a [First In First Out \(FIFO\)](#) structure built with registers. To handle inputs from two different sources, the scheduler maintains a separate [FIFO](#) queue for each source (i.e. one *new flows* queue and one *returning flows* queue) and alternates between the two queues when selecting the next flow. It is possible to replace each of these queues with priority queues if certain flows need to be prioritized over others.

Information Table. As explained in Section [2.2.3](#), the scheduler needs to keep track of two pieces of information in order to properly schedule the flows. One is whether the flow has an active event in [TPE](#) and another is whether the flow has any more events to

process. We track this information in a table consisting of two bits per flow: the first bits, which we will call “isActive”, indicates whether or not a flow has any event in either the scheduler or the [TPE](#). The second bit, called “hasNewEvents” tells if any new event for a certain flow has arrived since the last time it was set to zero.

The way the information in the table is used and updated is different for new flows and returning flows.

- **New flows.** when the new flow accesses the information table, it reads the table, if the flow has any events currently being processed (i.e., isActive is set), it sets the hasNewEvents bit to 1 and does not move to the next stages. Otherwise, it means this flow just got its first event or a new event after an idle period, and has not events in the processing pipeline. As such, isActive is set to 1 and the hasNewEvents bit is set to 0. The latter ensures that the scheduler can track the receipt of events for this particular flow from this point onward. Afterwards, it joins the safe flow queues in the next stages.
- **Returning flows.** When an event exits the scheduler and is sent to the [TPE](#) in the final stage of the pipeline, it carries a signal indicating whether it was the last remaining event in its flow queue. This information comes from the event memory, which maintains the status of all per-flow event queues, specifically, whether each queue is empty. If the queue becomes empty as the event leaves, the corresponding empty signal is used to set this flag. We refer to this signal as “wasLastEvent.”

When a flow returns to the pipeline from [TPE](#), it will access the table and make its decision based on the hasNewEvents and wasLastEvent bits. if either the hasNewEvent is 1, or the wasLastEvent is 0, it means that there are more events this flow has to process, in this case it will set the hasNewEvents to 0 and keep the isActive as 1 and join the safe flows again. This way, we can know if any new events arrived for this flow between this access and the next one. If hasNewEvents is 0 and wasLastEvent is 1, it means that there are no more events in the event memory for this particular flow, so it will set isActive to 0 and will not move to the next stages.

There may be cases where both the incoming flow and the returning flow have the same flow ID. If this happens, the table will detect the conflict and set both hasNewEvents and isActive bits to 1. and return this value instead of the value of the table for that particular cycle. This assignment corresponds to the scenario where the returning event arrived first and the returning event arrived right after, this way the information table remains consistent.

Event FIFO Memory. To keep track of the events that arrive at the scheduler, we implemented a per-flow **FIFO** memory. The overall design of this module is shown in Figure 2.5. Reading from and writing to this memory is performed through a two-stage pipeline. The per-flow buffers that keep the events are implemented as ring buffers with a read and write pointer, which will be updated after each read or write operation respectively.

When events arrive, the flow ID is used to access the write pointer memory. This memory represents the current position of the write pointer for each queue and is initialized to zero for all flows. Once the current position is obtained, the combination of this pointer and the flow ID determines the exact memory address for writing the event. In the second stage of the pipeline, the event is written to memory, and the write pointer is incremented by one and stored in the corresponding memory location in the write pointers memory.

Once the scheduler decides which flow should send an event to the **TPE** next, it will have to read the event from the event buffer memory. To do so, it first accesses the read pointer memory, which tracks the current read position for each queue and is also initialized to zero. Similar to the write path, the combination of this pointer and the flow ID identifies the memory address for the next read in the event buffer memory. In the second stage of the pipeline, the event is read, the read pointer is incremented by one, and the read pointer memory is updated accordingly.

To track the status of each queue, specifically, whether it is full or empty at any given moment, we maintain a separate counter memory called the *Item Counter*. This memory records the number of entries in each queue and is updated after every read or write. After each update, the values are used to refresh the empty and full bit vectors, which contain one bit per flow indicating its current status. Without this memory, the scheduler would need to access both the read and write pointer memory and compare them to determine the status of the flow. However, both memories have only one read port that is being used by either the read or write operation but not both. This method would require them to either have two read ports to allow both operations to read from them, which would need us to maintain a copy of each memory. Or for the pipeline to take extra cycles to allow each memory to be accessed twice. Therefore, using the *Item Counter* enables efficient tracking of queue status without introducing extra pipeline cycles or memory modules.

The “Event Memory”, which is responsible for storing all pending events, is implemented as a dual-ported **BRAM**. The other memories are synthesized to registers, but depending on the parameters of the design, they can also be implemented as a **BRAM** if their size is large enough.

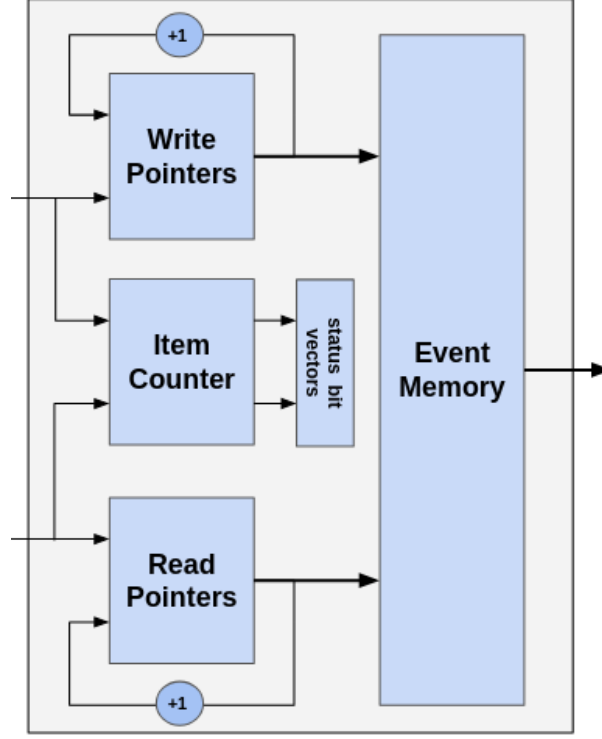


Figure 2.5: The hardware design for the event FIFO memory.

2.3 Handling Instructions

The **TPE** can issue requests for various actions to be carried out by different modules in the **TPS**. These requests are given to **TPS** in the form of instructions. Each instruction encodes the specific operation that an execution module should perform, and its structure varies depending on the destination module. There are three different instruction handlers in **HtraP**: Packet Generator, Reassembly module and Timer. Table 2.2 provides a complete list of the instruction types supported in **HtraP** together with their corresponding target modules. The exact structure of each instruction will be explained in the following subsections.

Timer. The timer module is responsible for tracking a predetermined number of timers for each flow. As input, the timer receives an event along with a duration and will send that event to the scheduler when the timer expires. Each timer's associated event is stored in a per-flow *Timer Event Memory*. Each flow can request one of three operations on a

timer: (1) Start: to start a new timer with the given duration and generate the given event upon timeout, (2) Restart: to restart the timer with the given duration, and generate the previously stored event upon timeout, or (3) Cancel: to stop the timer from generating an event.

To support multiple flows, the timer module stores the received durations in a per-flow memory referred to as the *Timer Duration Memory*. The timer iterates over this memory, checking one entry per clock cycle. If a timeout is detected, it retrieves the corresponding event from the *Timer Event Memory* and sends it to the scheduler. Since the module checks only one flow per cycle, there may be a slight delay in detecting some timeouts. However, for timers on the order of milliseconds or longer, this delay is negligible. This design may not be suitable for finer-grained timers requiring higher precision.

Our prototype does not include the timer module. However, similar per-flow timer modules have been implemented in prior work, such as Tonic [7], from which our design draws inspiration. We believe this provides sufficient evidence to support the feasibility of our proposed design.

Target	Instruction	Description
Packet Generator	generate packet	Instructs the module to fetch the corresponding data from memory and generate a packet. Can also handle segmentation.
	set scheduling parameter	Updates the packet scheduling parameter of a certain flow if needed.
Reassembly module	allocate buffer	allocates a new data buffer to the flow.
	add data segment	adds the segment to the specified location in the buffer.
	flush and notify	Makes the reassembled data available to the application.
Timer	start timer	sets a new timer to create an event upon expiration
	restart timer	restarts an already created timer
	cancel timer	cancels the timer

Table 2.2: A list of the instructions in [HtraP](#)

2.3.1 Packet Generator

Upon receiving a request, it first fetches the payload from the specified memory location and stores it in a temporary buffer. Once the data is loaded, the module appends the appropriate header fields and generates the final packet to be sent over the network.

In addition to basic packet generation, users can instruct the packet generator to create multiple segments from a single large data segment. For example, in the TCP protocol, the user may request the packet generator to segment the data into multiple packets, each with a fixed size, and to update the sequence number in each packet based on the length of the previous segment. To support this behavior, the packet generator includes a configurable header update policy that can automatically adjust fields like the sequence number across segments. Furthermore, to ensure fairness among multiple multi-segment instructions from different flows, the next segment to be generated is selected in a round-robin fashion among them.

Instruction

Packet Generation. The packet generation instruction is made up of several key fields, each of which provides essential information for the packet generator to correctly create packets. These fields and their descriptions are as follows:

- **Data Address:** This field indicates the address in memory where the payload is located. The packet generator uses this address to fetch the data for packet construction.
- **Data Length:** This field specifies the total length of data to be loaded from memory. If this value is set to zero, the packet generator will skip the data-fetch step and proceed to create the packet without any payload.
- **Segment Length:** This field defines the amount of data to be included in each individual packet. If the segment length is smaller than the data length, the packet generator will perform segmentation to divide the loaded data segment into multiple packets.
- **Update Value:** This field provides the value that should be used to update the specific header fields of the packet during segmentation. The packet generator includes a configurable module that determines how this value should be applied. The details of this module will be explained later in this subsection.

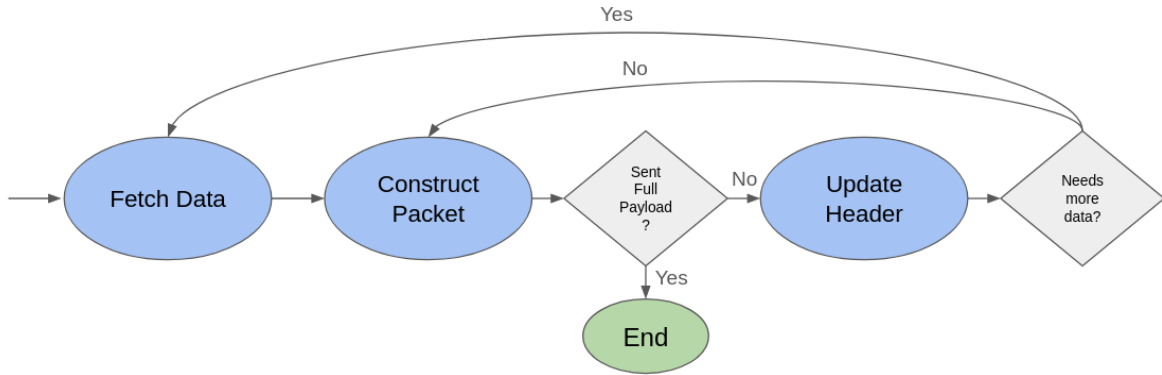


Figure 2.6: The packet generator algorithm

- **Header:** This field contains the header to be used in the generated packet. In the case of segmentation, this header applies to the first packet, and subsequent packets will have their headers updated using the update value and the configuration specified by the user.

Packet Scheduling. The packet scheduling instructions control the packet transfer scheduling parameters. Depending on the scheduling algorithm, they can represent either a grant-based or a rate-based system.

- **Parameter Type:** This field indicates the type of the scheduling parameter.
- **Parameter Value:** This field specifies the value for the scheduling parameter. This can either be the transfer rate or token count.

Logical Diagram

Figure 2.6 illustrates the general logic of the packet generator.

The packet generator can be divided into three phases: data fetch, packet construction, and header update.

Fetch Data. Upon arriving at the packet generator, instructions are assigned an instruction ID and sent to the fetch data module. This module, implemented as a state machine, retrieves data from where it is stored and stores it in a per-instruction temporary

FIFO buffer. It continues loading data until either the buffer is full or all data has been loaded. After loading the data, the instruction proceeds to the next phase.

Packet Construction. After the data is loaded into the buffer, this module reads X number of bits, where X is the segment length specified in the instruction, from the **FIFO**. Afterwards, it attaches the header to the payload, and sends the packet out. The instruction is then passed to the next stage.

During this process, we track the amount of data sent so far. After each packet is sent, we check whether all data requested in the instruction have been transmitted by comparing this value with the data length specified in the instruction. If all data have been sent, the packet generation for this instruction is complete; otherwise, a new header must be generated for the next data segment.

Header Update. In this phase, the header is updated to reflect the expected header for the next packet. This step is user-configurable and can be set to generate the next header based on the current header's value. For example, in TCP, this phase would be configured to increment the sequence number by the length of the previous packet's payload.

After updating the header, we verify if enough data is available in the instruction's temporary data buffer for the next packet. If not, the data address and data length are updated, and the instruction is sent back to the fetch data phase to retrieve more data. If sufficient data is available, the instruction proceeds to the packet construction stage to send the next segment.

Hardware Design

In order to execute the described algorithm, the packet generator module is implemented as a pipelined hardware block that receives instructions from the **TPE** and generates corresponding network packets. Its architecture is designed to support high-throughput packet transmission while maintaining flexibility for user-defined behaviors such as segmentation and header updates.

The generator is organized into several hardware stages corresponding to the ones mentioned in Figure 2.6.

The payloads of packets are often stored in external memory, which typically has limitations on read and write efficiency. To address this, we read a large chunk of data at once and store it in a temporary memory that allows faster access during individual packet generation. Temporary payload data is buffered inside a **FIFO** buffer, allowing the packet

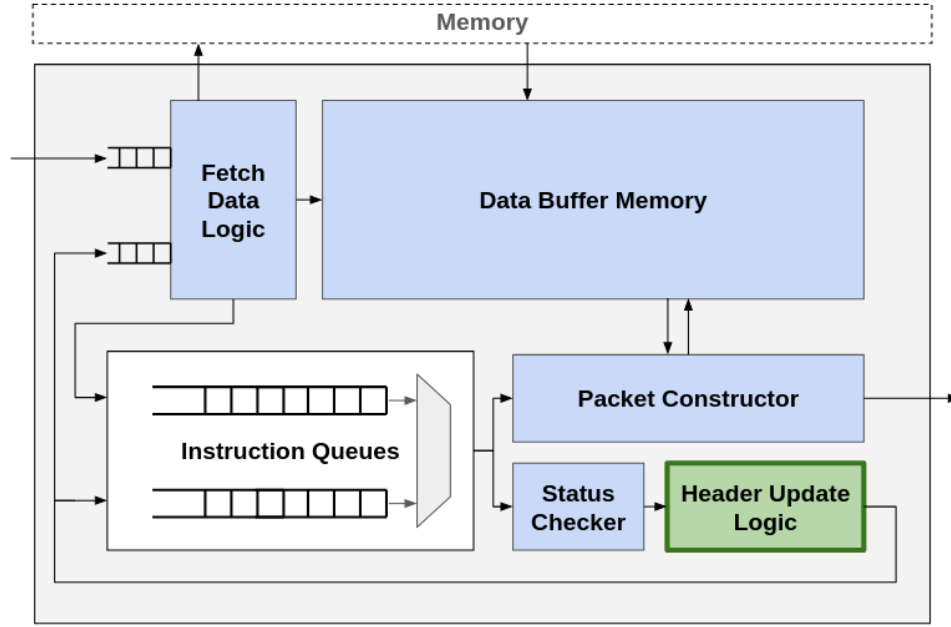


Figure 2.7: The packet generator design. The green box is programmable

constructor to access data efficiently while new data is fetched in parallel. Each instruction has access to its own [FIFO](#) buffer to manage its data segments.

The rest of this subsection will describe the implementation details of each stage in the packet generator pipeline, along with the data structures and interfaces used to support its operation. The design for the packet generation module can be seen in figure [2.7](#).

Data Buffer Memory. The data segments needed by the instructions is temporarily stored in the data buffer memory. This buffer consists of a set of [FIFO](#) memories, where each instruction has access to one [FIFO](#) buffer for reading and storing its data segments. The implementation of this memory is similar to the scheduler’s event [FIFO](#) memory shown in Figure [2.5](#), but instead of events, each [FIFO](#) will store the data inside the memory in 64 byte chunks. Although the size of each element inside the buffers is set to 64 bytes, it can be adjusted as needed. The depth of the buffers is also configurable and can be tuned based on the requirements of the application.

Fetch Data Logic. The first stage of the packet generator is responsible for fetching the payload data associated with each instruction. Incoming instructions are initially stored in a queue and are processed in order. An instruction leaves the queue only after the previous instruction has completed its data loading phase. The data loading process

itself is implemented as a simple state machine that continues reading data into a per-instruction buffer until either the buffer reaches its capacity or the entire payload has been loaded. Once the required data has been fetched, the instruction is forwarded to the next stage of the packet generator for further processing.

Instruction Queues. The instruction queues are where instructions are stored before being processed by the packet constructor. An instruction joins the queue after initiating the fetch process. To ensure fairness between different flows, instructions that need to generate multiple packets via segmentation must wait after generating one packet, allowing other flows to generate their packets as well. These instructions can then generate their next packet afterwards. To enforce this policy, each instruction generates exactly one packet upon leaving the queue. If an instruction uses segmentation and needs to produce multiple packets, it re-enters the instruction queue after receiving updates to its relevant fields, leading to a round-robin between different instructions.

Packet scheduling policies are applied through this module, where users can control packet consumption using rate limiting or grant tokens. For the prototype discussed in Chapter 4, we implement a [FIFO](#) queue as described, and do not support scheduling for each individual flow. However, future implementations could take inspiration from Tonic[7] to support more sophisticated packet scheduling for each flow.

Packet Constructor. The packet constructor is responsible for constructing and transmitting the final packet. It receives the header from the instruction and reads the corresponding payload data from the data buffer. These two components are combined to form a complete packet, which is then sent to the network interface. Once the packet has been successfully transmitted, the constructor signals the instruction queue to release the next instruction.

Status Checker. Once an instruction is sent to the packet constructor, the status checker evaluates whether additional packets need to be generated and whether more data must be fetched. The first step is to update the amount of data sent so far. This is done by adding the size of the current payload segment, which is determined as the minimum of the configured segment size and the remaining data to be transmitted.

After updating this value, the status checker computes whether any data remains to be sent. If the remaining data are zero, the instruction is considered complete and no further action is required for that flow. However, if data remain, the module proceeds to inspect the current state of the data buffer. In cases where the entire data segment has not yet been loaded and the available data in the buffer falls below a predefined threshold, the instruction is marked to be sent back to the data fetch stage after the header is updated. By default, this threshold is set to zero, meaning that the instruction will only be sent to fetch

data after it has used up all of its available data, but this value can be tuned for different application needs. Once the status updates are complete, the instruction is forwarded to the header update module, which will prepare the header for the next segment.

Header Update Logic. This module updates the header field values in the instruction for the next segment. It is user-configurable and can be tailored to perform the desired updates. The header update module has access to all fields of the instruction—specifically, the data address, data size, segment size, update value, and current header. The update value is a 32-bit field that can be set by the [TPE](#) and used within this module to assist in header modification. While the size of this input can be adjusted as needed, the total size of the instruction should not exceed 64 bytes.

In addition to the instruction fields, the module also receives information about the total number of bits transmitted so far. This can be useful for determining the size of the next segment, if necessary. Listing 2.2 provides an example of header modification in the context of TCP segmentation, where the header is updated by adding the length of the previous segment.

After updating the appropriate header fields, the packet generator will insert the updated instruction back into the instruction queues.

```
always@(posedge clk)begin
    if(rst)begin
        Header_out <= 0;
    end
    else begin
        Header_out[SEQNUM_FIELD_OFFSET-1:0] <= Header_in[
            SEQNUM_FIELD_OFFSET-1:0];
        Header_out[SEQNUM_FIELD_OFFSET+SEQNUM_FIELD_SIZE-1:
            SEQNUM_FIELD_OFFSET] <= Header_out[SEQNUM_FIELD_OFFSET+
            SEQNUM_FIELD_SIZE-1:SEQNUM_FIELD_OFFSET] + Segment_size_in;
        Header_out[HEADER_SIZE-1:SEQNUM_FIELD_OFFSET+SEQNUM_FIELD_SIZE]
            <= Header_out[HEADER_SIZE-1:SEQNUM_FIELD_OFFSET+
            SEQNUM_FIELD_SIZE];
    end
end
```

Listing 2.2: Header Update Example

2.3.2 Reassembly

The reassembly module handles the process of putting data segments back together into full messages before making them available to the application. To do this task, it will keep the data segments in an internal memory we will refer to as the *Reassembly Data Buffer (RDB) Memory*. It supports three types of instructions: “allocate buffer,” “add data segment,” and “flush and notify.” When it receives an “allocate buffer” instruction, it reserves one of the buffers in the *RDB* memory for the flow. With an “add data segment” instruction, it places the new segment into the right spot in the buffer based on the given offset. Finally, after receiving a “flush and notify” instruction, it makes the data available to the application by sending moving it to a application-accessible memory. This module does not track the state of the segments it receives. Instead, it will rely on the *TPE* to provide it with the correct offset for insertion and flush for data transfer, and merely inserts the event in the location specified by the instruction.

Allocate buffer. This instruction will reserve a buffer for the given flow id. The inputs for this instruction is as follows:

- **Flow ID:** This field represents the identifier of the flow requesting a buffer.
- **Buffer ID:** This field represents the *TPE*-assigned ID of the buffer for the flow. The *TPE* assigns these IDs to differentiate between its reserved buffers and send requests for them.

Add data segment. This instruction asks the module to fetch a piece of data from the the *Temporary Data Memory* where the packet parser temporarily stores the payloads of incoming packets and insert it into its correct place in the buffer. The exact offset of the insertion will be given to the module as input, therefore, the responsibility of tracking the segment order falls on *TPE* and the reassembly module is only supposed to carry out the instruction. The fields of this instruction are as follows:

- **Flow ID:** The identifier of the flow making the request.
- **Buffer ID:** *TPE*-assigned ID of the buffer for the flow.
- **Address:** The memory location where the current data segment (coming from a recent packet payload) is stored.
- **Length:** The size of the data segment.

- **Offset:** The position within the buffer where the data should be inserted.

Flush and notify. This instruction asks the reassembly module to send the buffer up to a certain point to the application. Similar to the "add segment" instruction, the calculation of the exact size of the transfer falls on the [TPE](#), and the reassembly module is only responsible for sending the data to memory. The instruction has the following fields:

- **Flow ID:** The identifier of the flow making the request.
- **Buffer ID:** [TPE](#)-assigned ID of the buffer for the flow.
- **Address:** The address in the application-accessible memory where the data will be transferred.
- **Length:** The size of the data being flushed.

Hardware Implementation

This hardware design for the reassembly module needs to be able to execute the three aforementioned instructions. Figure 2.8 shows the design of this module. In this design, the flush and notify instruction can run in parallel to the add data segment instruction.

After arriving at the reassembly module, the instructions will access the buffer ID mapper to get the reassembly module's local buffer ID based on the request. This local buffer ID is used by the reassembly module to identify the [RDB](#) the instruction is referring to. At this stage, the "allocate buffer" request will be executed by modifying the contents of the "buffer ID mapper". We treat the buffer ID mapper as a memory accessed using the combination of the flow ID and the [TPE](#)-assigned buffer ID from the request.

After this stage, the "flush and notify" and "add data segment" instructions will be inserted into the instruction queues and take separate paths.

Allocate Buffer. This instruction assigns a local buffer ID to the flow using the combination of the flow ID and the [TPE](#)-assigned buffer ID. The local buffer ID to be assigned is determined using a counter that increments with each assignment.

Add Data Segment. This instruction retrieves the segment data from the temporary data memory in the parser and inserts it into the [RDB](#) memory using the offset specified in the instruction. The offset indicates the exact byte position where the data should start. Because the [RDB](#) is divided into 64-byte chunks, we may need to shift the incoming chunks before storing them in [RDB](#). For example, consider a scenario where the offset value is set

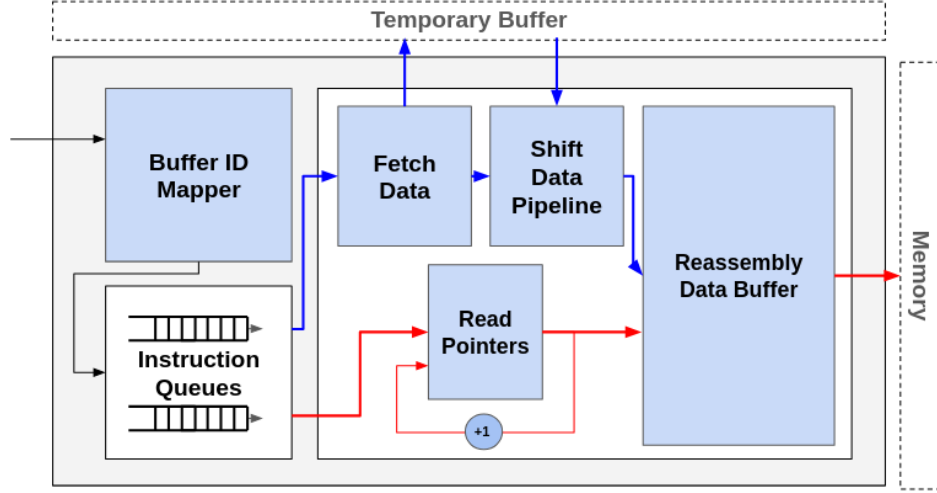


Figure 2.8: The design of the reassembly module. The red and blue arrow represent the path for “flush and notify” and “add data segment” instructions respectively

to 16. This means that the data chunk should be stored starting from the 16th byte of the [RDB](#). However, since the [RDB](#) is divided into 64-byte chunks, we must store the first $64 - 16 = 48$ bytes of the chunk in address 0 of the [RDB](#), and the remaining bytes in address 1. Moreover, the first 384 bits must be placed at the end of the chunk in address 0. Figure 2.9 illustrates this scenario for a 256-byte data segment. To meet this condition, the data is shifted before being inserted into the [RDB](#) memory. This shifting is handled by the *Shift Data Pipeline*.

- Fetch Data.** This component will fetch the data from the temporary data memory in 64 byte chunks. While this component is working, the other add data segment instructions would need to wait because the fetch process may take more than one cycle, and the temporary data memory has only one read port. After fetching each chunk, it will be sent to the *shift data* component. Figure 2.9 shows the data after it has been shifted. Once all data has been fetched, the component must wait one cycle before starting the next instruction. This is because, after the segments are shifted, an additional cycle is required to store the final piece of data. Figure 2.9 illustrates this scenario: writing the 4-chunk data segments takes 5 cycles, as the shift introduces an extra segment for writing into the [RDB](#).
- Shift Data Pipeline.** This component is a 6-stage pipeline that shifts the data

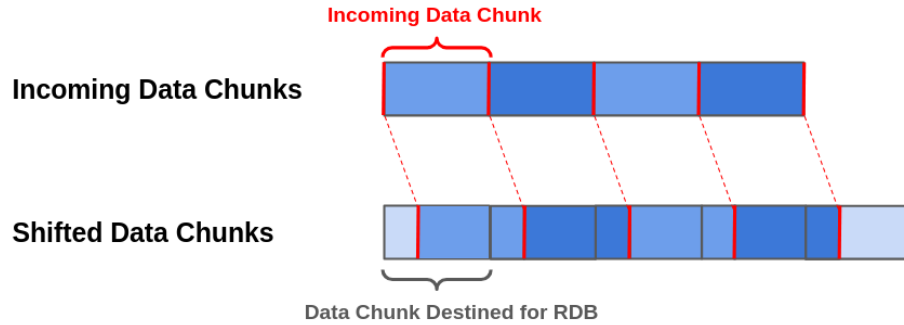


Figure 2.9: Adding a 256 byte data segment to RDB

according to the offset value. The first 6 bits of the offset are used to determine the shift amount, and the remaining bits specify the location of the data chunk in the RDB. To perform the shift, each stage of the pipeline processes a specific shift value: the i th stage examines the i th bit of the shift amount and shifts the data by 2^i bytes. When a segment contains multiple chunks, starting from the second chunk, the pipeline concatenates the previous chunk with the current one, shifts them together, and then selects the first 64 bytes of the result. This ensures that any portion of the previous chunk that needs to be moved into another chunk is correctly accounted for. The result of the shift is then stored in the location specified by the remaining bits of the offset. For the first and last chunks of the incoming data, it is possible that other segment values are already stored in memory. For example, we may have received a 256-byte segment at offset . To handle this, the component reads the memory for these two locations, concatenates the pre-existing values with the shifted segment, and stores the final value back into the buffer.

Flush And Notify. With this instruction, the reassembly module sends a portion of the specified RDB to the application-accessible memory. The instruction specifies the size of the data to be transferred. A read pointer is maintained for each buffer in the *Read Pointers* memory. To transfer the data, the module reads from this memory and updates the pointer until the requested size has been transferred. It is important to note that the specified size is expected to align with the 64-byte chunk boundaries.

Chapter 3

Transport Processing Engine

[TPE](#) is the programmable part of [HtraP](#) and where the main logic of the protocol is located. The [TPE](#) is implemented as a series of [HLS](#) programs that will receive the event and state of the flow as input, do the relevant processes, update the state variables and generate instructions if needed.

3.1 Programming in TPE

The programming for the [TPE](#) is done in [HLS](#). [HLS](#) is a tool that allows developers to describe hardware designs using high-level programming languages such as C and C++ instead of hardware description languages like Verilog. The [HLS](#) tool will receive these programs and translate the high-level code into [RTL](#) implementations that can be synthesized onto [Field-Programmable Gate Arrays \(FPGA\)](#)s. The syntax and capabilities may vary across different HLS tools. For [HtraP](#), the interface is designed to be synthesized using the Vitis HLS [\[2\]](#) tool.

Inside the [TPE](#), the user defines a series of functions that describes how a particular event should be processed. These functions, which we will refer to as “Event Processors”, form the core logic of the [TPE](#). Each Event Processor specifies how the associated event interacts with and updates the context of its flow and what instructions it should generate as a result. There must be exactly one Event Processor for each defined event type. The modular structure of the interface allows the programmer to easily add new protocols or modify existing ones by updating the corresponding event processor.

3.1.1 Event Processor Interface

The event processors are implemented as a C++ function. All functions should be implemented using the given interface, although slight variations are allowed. For example, the number or type of the generated instructions may vary across different processors. Even if the same syntax is used, the [HLS](#) tool might remove the interface that was not used during synthesis. The interface of an event processor is given in [Listing 3.1](#).

```
typedef hls::stream stream;
void X_event_processor (stream<X>& event_in,
    stream<ctx>& ctx_in, stream<ctx>& ctx_out,
    stream<pktGen_instr> p[N], stream<reasm_instr> r[M],
    stream<timer_instr> t[K], hls::stream<event>& flow_out)
```

Listing 3.1: Event processor interface

All inputs and outputs of the module are defined with the `hls::stream` type. Once synthesized, this type will be implemented as an AXI4-Stream interface.

In [Listing 3.1](#), `X` is the event type, `event_in` and `ctx_in` are the input streams, and the rest are output streams. `M`, `N`, and `K` are constants representing the number of instructions the event processor will generate for each type. the `flow_out` is the information from the incoming event that must be passed on to the scheduler. Note that even though the interface does not explicitly indicate whether a stream is an input or an output stream, its usage within the code determines its type. Therefore, it is important to respect the type while writing the program. The method of using the input and output streams within the program will be described in [Section 3.1.2](#).

The rest of this section will explain the elements of the interface in more detail.

Event and Context

The input of the event processor includes the event and the context, both of which will provide the information necessary to implement the protocol logic.

Events. The events are given to the program in the `stream <X>& event_in` input. As explained in [Section 2.2.1](#), The events are abstractions that represent different occurrences within the transport layer program. The exact type of this stream's element is determined by the type of the event this function is going to process.

Each event type will be defined as a C struct, and the fields of the event represent the protocol-relevant metadata carried by this event. In addition to these metadata, the event also carries data that would help the [TPS](#) modules make correct decisions. This data includes the flow ID and the notEmpty bit, Both of which will be used by the scheduler to determine the state of the flow. The exact role of these bits is explained in [Section 2.2.3](#). The flow ID will also be used to update the context. These two fields are included in the `event` struct, which is the parent type that all event types should inherit from. Every event processor should write its incoming event to the `flow_out` stream to be used in the next stages.

One example of an event is given in [Listing 3.2](#)

```
struct event{
    bool has_more_data;
    ap_uint<FLOW_ID_WIDTH> flow_id;
}

struct ACK: public event {
    ap_uint<32> ack_seq;
    ap_uint<32> rwnd_size;
};
```

Listing 3.2: An example of an event struct

The fields of the event can be of any C++ primitive type, or the `hls::ap_uint<N>` and `hls::ap_int<N>` types, which indicates an N-bit number.

Context. The context is the per-flow state of the program, and is given to the event processor through the `stream < ctx >& ctx_in` as input, and the updated state will return to the context memory through the `stream < ctx >& ctx_out` interface. Similar to the event, the `ctx` type is defined as a C struct. All fields are defined by the programmer and can have any type supported by the Vitis HLS tool.

The size of the context is an important consideration in the design. Since the context carries most of the information related to the event processor, it is likely to be large. The size of this struct will affect the size of the per-flow context memory. Therefore, it is important to optimize the context size.

Packet Generation Instructions

This instruction is sent to the packet generator for processing, therefore its fields match the fields of the packet generation instruction explained in [Section 2.3.1](#), which includes

Data Address, Payload Length, Segment Length, Update Value and Header. Listing 3.3 shows the built-in C struct that represents a packet generation instruction.

```
struct pktGen_instr {
    ap_uint<32> addr;
    ap_uint<32> payload_len;
    ap_uint<32> segment_len;
    ap_uint<32> update_val;
    Header header;
    ap_uint<16> header_size;
};
```

Listing 3.3: Packet generation instruction type

Header. The `Header` type in `pktGen_instr` represents the header of the packet, which as explained in Section 2.3.1, will be directly used to generate the header of the first packet segment and will serve as a basis for generating the headers of the future segments. It is possible to define this header as a struct whose fields represent the fields of the final header. However, some protocols such as [RoCEv2](#) may have different headers for different types of packets. In this case, the size of the `Header` type should be the maximum size of these headers, with shorter headers being extended at the end to match this size. In this case, the `header_size` field will be used to indicate the actual size of the smaller headers to the packet generator.

Data Reassembly Instructions

The fields of this instruction matches the fields of the reassembly instructions explained in Section 2.3.2. The reassembly module has three different instructions, but all of them use the same struct, and the type of the instruction should be specified by the programmer when creating it. Listing 3.4 shows the built-in C struct that represents a reassembly instruction.

```
struct reassem_instr {
    ap_uint<2> instr_type;
    ap_uint<FLOW_ID_WIDTH> flow_id;
    ap_uint<4> buffer_id;
    ap_uint<32> addr;
    ap_uint<32> data_length;
```

```
    ap_uint<32> data_offset;
};
```

Listing 3.4: Packet generation instruction type

The `instr_type` represents the type of the instruction, which can be either one of the following: (1) `ALLOCATE_BUFFER`, (2) `ADD_DATA`, or (3) `FLUSH_AND_NOTIFY`. The first type only needs to use the first two fields. The second one will use all fields and the third one use all fields except the `data_offset`. The names of these fields matches the ones explained in Section 2.3.2, and they provide the same information to the reassembly module.

Timer Instructions

The instructions heading to the timer module can have three different types: (1) start, (2) restart, and (3) cancel. All of these instructions will be represented by a single `timer_instr` type. The fields of this type is shown in Listing 3.5

```
struct TimerInstr {
    ap_uint<2> cmd;
    ap_uint<LOG2_OF_PER_FLOW_TIMER_COUNT> timer_id;
    ap_uint<32> duration;
    ap_uint<2> unit;
    TimerEvent event;
};
```

Listing 3.5: Timer instruction type

The `cmd` field represents the command of the instruction, which would either be 0 (Start), 1 (Restart) or 2 (Cancel). The `timer_id` represents the specific timer the instruction should modify. Each flow has access to a limited number of timers that are predefined by the user; therefore, the `LOG2_OF_PER_FLOW_TIMER_COUNT` should be set by the programmer as well. The `duration` represents the duration of the timer. This value will only be set for the start and restart instructions. The `unit` field represents the unit of time. The number is mapped to one of the following: nanosecond, microsecond, millisecond, or second. Finally, the `event` field represents the event that should be generated by the timer upon expiration.

TimerEvent. This type represents the type of `event`. If the timer only generates one type of event, this value can be defined as that type. In the case of multiple events, the

size of the `TimerEvent` should match the size of the largest event the timer is going to generate.

3.1.2 Event Processor Implementation

The body of an event processor contains the protocol response to a single event. In general, the program will first read the inputs from the input streams, perform some operations, and write the outputs to an output stream. This process is similar across all event processors. Listing 3.6 shows an example of this process.

```
void data_ep (hls::stream<X>& ev_in, hls::stream<context>& ctx_in, hls::
stream<context>& ctx_out, hls::stream<PktGen_instr>& PktGen_instr_out,
hls::stream<sched_info>& flow_out) {
    if(!ev_in.empty() && !ctx_in.empty()){
        X ev;
        context ctx;

        ev = ev_in.read();
        ctx = ctx_in.read();

        PktGen_instr pgi;

        /*Protocol Logic*/

        event event_flow_info;
        event_flow_info.flow_id = ev.flow_id;
        event_flow_info.has_more_data = ev.has_more_data;

        flow_out.write(event_flow_info);
        ctx_out.write(ctx);
        PktGen_instr_out.write(pgi);

        return;
    }
}
```

Listing 3.6: An example of an event processor

In this section, we will discuss some of the challenges involved in writing [HLS](#) programs for the [TPE](#).

Loops. In order to pipeline the program, [HLS](#) will unroll any loop inside. Therefore, it is important for the loops in the program to have a fixed number of iterations, and

unbounded loops are generally not supported. Additionally, the loops where each iteration depends on the result of the previous one may end up creating long pipelines. While our design is able to handle such pipelines, it is better to avoid longer pipelines when possible to keep the latency low. Therefore, we need to optimize the program to avoid such loops as much as possible.

Arrays. While it is possible to have arrays inside the context, the size of the context usually grows considerably with any increase in the number of elements inside the array. Having a large context cause problems with the [HLS](#) interface, and it usually would need to be divided into smaller parts. Furthermore, having a large input move across several pipeline stages can negatively affect the latency of the design. Therefore, it is important to make sure that the size of the context does not grow too large. However, this may cause problems with some implementations, for example, [RoCEv2](#) needs to keep a list of all pending requests. For our implementation, we have optimized the size of these queues to a value that would be reasonable for the program, but may not be optimal.

3.2 Interface With [TPS](#)

As explained so far, the [TPE](#) is implemented as a set of different event processing functions. In order to integrate these event processors in [TPS](#), we need to be able to guide an incoming event to its correct processor. Additionally, we need to collect the outputs of these modules and send them to their correct handler. For this thesis, we implemented this manually, but in a future version of [HtraP](#), this part can be auto-generated by a dedicated compiler, since this section follows a fixed set of steps.

Incoming Event Interface. As mentioned in Section [2.2.1](#), the incoming events carry some additional bits with themselves that indicates their type. In order to connect an event to [TPE](#), we first need to choose an ID for each event. Then, we assign the ID of the event to its corresponding event processor, and use a de-multiplexer to connect the incoming event to its event processor.

Outgoing Context and Instruction Interface. The [TPE](#) should output at most one instruction of each type every cycle, and should also send out one context back to the context memory. Since we are allowing the different event processors to have different cycle lengths, it may be possible for some event processors to generate an output at the same time. In these cases, the outputs should be stored in a queue and be sent out one by one. Therefore, each event processor would need to have a small wait queue to store its outputs in case of a conflict, and we can select from these queues in a round-robin fashion.

If no conflict is present, there is no need to store the value in the queue and it may be sent out directly. Because these queues are only used when simultaneous outputs occur and are typically short, their impact on the overall system latency is minimal.

For instructions, additional considerations are needed. As discussed in Section 3.1, event processors may generate multiple instructions at once. However, since the handler modules can only process one instruction per cycle, these instructions must be sent sequentially. As a result, some event processors must aggregate their instructions in a queue before transmission.

3.2.1 Configuring the Parameters

As mentioned in section 2.1, there are a number of parameters in [TPS](#) that can only be set after the implementation of [TPE](#). Table 3.1 explained how each parameter should be set based on the [TPE](#) implementation.

Module	Parameter	Value
All of the Design	EVENT_WIDTH	Maximum width of the defined events.
	EVENT_TYPE_COUNT	Total number of the defined events.
	CONTEXT_WIDTH	Width of the defined context,
Packet Generator	HEADER_WIDTH	maximum size of the defined headers.

Table 3.1: Parameters configured to match [TPE](#)

Chapter 4

Evaluation

In order to evaluate the effectiveness of our design, we implement a prototype in Verilog in 3540 lines of code and synthesized the design for the Alveo U250 data center accelerator card. In this chapter, we establish the hardware feasibility of our design in section 4.1. Section 4.2 provides microbenchmarks for different components of [TPS](#), including the scheduler, packet generator, and reassembly module. Section 4.3 will present implementations of different protocols in [TPE](#) to establish the expressiveness of the mostly protocol-independent interface between [TPE](#) and [TPS](#) across different protocols, and finally, Section 4.4 will evaluate a complete [TPE](#) and [TPS](#) co-design of the [TCP](#) protocol to establish the possibility of implementing a full transport protocol in [HtraP](#).

4.1 [HtraP](#) Hardware Feasibility

To evaluate the efficiency of our hardware design, we will look at two main criteria: (1) resource usage and (2) timing.

Resource usage. When the Verilog program is synthesized, it will be mapped to the hardware resources in the [FPGA](#) device. The primary resources are Registers, [Look-Up Table \(LUT\)](#)s, and [BRAMs](#). The [BRAMs](#) are used for local memories inside each module. Since the number of resources available in the [FPGA](#) are limited, it is important for the design to be able to fit into a given [FPGA](#) hardware.

Timing. Most of the modules in [HtraP](#) are designed as a multistage pipeline, with the memory access logic in the packet generator and the reassembly module being the only exception. In a pipelined hardware design, each stage of the pipeline will receive the input

Module	LUT	Registers	BRAM
Packet Generator	392 (< 1%)	1303 (< 1%)	2.5 (< 1%)
Scheduler	19066 (1.1%)	5728 (< 1%)	2.5 (< 1%)
Context	29 (< 1%)	0	8 (< 1%)
Reassembly	6855 (< 1%)	9671 (< 1%)	912.5 (34%)

Table 4.1: [HtraP](#)’s resource utilization for synthesis using the parameters in Table 4.2.

at the beginning of the stage, and must calculate the outputs of its stage within one clock cycle. In order for our hardware design to be feasible, we must ensure that it can meet the timing requirement of 4 ns. We choose this clock frequency because it allows us to keep-up with 100 Gbps line rate.

It is important to note that both of the mention criteria are heavily dependent on the type of the [FPGA](#) hardware, since a stronger [FPGA](#) is more likely to meet timing requirements, and also has more resources available to it. In our case, we choose the Alveo U250 data center accelerator card as the target device of our synthesis. The resource utilization report of our prototype for each module can be seen in Table 4.1.

As mentioned in Section 2.1, The [TPS](#) has various parameters that can affect the resource utilization of the different parts of the design. A report of the values used for this synthesis is presented in Table 4.2. For the [TPE](#) defined ones, we set them to match the [TPE](#) implementation of [TCP](#), which will be explained in more detail in Section 4.4. This was done so that we can verify the hardware feasibility of [HtraP](#) for a fully implemented protocol. The [NIC](#) assigned values were set based on the interface of OpenNIC [1], which is an open source project that provides a programmable [NIC](#) platform to the users. The rest of the values were set to default values.

4.2 Micro-benchmarks for the [TPS](#) Modules

4.2.1 Scheduler

When designing the scheduler, our primary objective was to achieve high throughput while ensuring memory consistency across all operations. To evaluate the effectiveness of our design, we conducted a series of measurements focused on both latency and throughput. For this purpose, we randomly generated several sequences of events over 50,000 cycles simulating a variety of traffic patterns. The outputs produced by the scheduler during these

Module	Parameter	Value
All of the Design	FLOW_COUNT	2048
	EVENT_WIDTH	64 b
	EVENT_TYPE_COUNT	4
	CONTEXT_WIDTH	938 b
	SERIALIZED_DATA_WIDTH	512 b
	SERIALIZED_PACKET_WIDTH	512 b
Scheduler	PER_FLOW_MEMORY_DEPTH	16
	EVENT_QUEUE_DEPTH	8
Packet Generator	HEADER_WIDTH	168 b
	INSTRUCTION_QUEUE_DEPTH	8
Reassembly module	PER_BUFFER_MEMORY_DEPTH	512

Table 4.2: The list of the synthesis values for [TPS](#) parameters.

cycles were recorded and analyzed to measure its performance under different operating conditions.

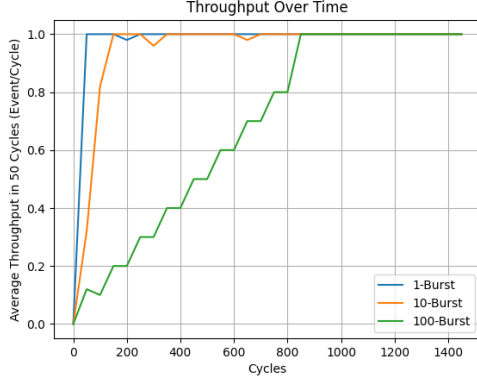
Event per Cycle Throughput

One goal of the scheduler is to output as many events as possible in each clock cycle. Although this number depends on the traffic pattern, we expect it to approach one event per cycle over time. The factors that can affect throughput include the pipeline length and the characteristics of the flow traffic.

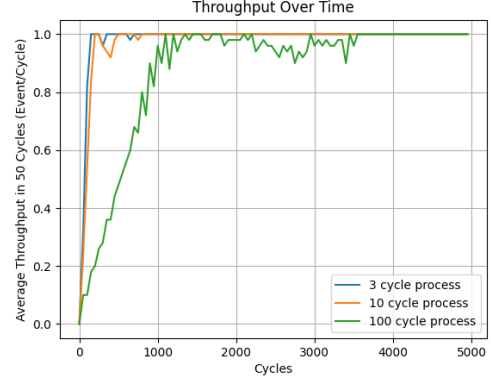
To evaluate the effectiveness of our design, we measure the scheduler’s throughput under different input patterns. Two important aspects of the traffic pattern are the number of active flows and the burstiness of the input traffic.

Number of active flows. If the number of active flows is small, throughput is likely to be low, as the scheduler may not find a safe event to send to the [TPE](#). However, we observe that as long as the number of active flows significantly exceeds the pipeline length, this factor does not impact throughput. Since our design is intended to support a large number of active flows on average, we expect this factor to have minimal effect.

Bursty traffic. We define a flow as bursty if it receives multiple consecutive events in a short period of time. In such cases, the scheduler cannot dispatch new events from that flow until its previous events have exited the pipeline. This can reduce throughput if no other events are available to schedule. To test this effect, we consider an extreme case: an



(a) The throughput over time for the TPE pipeline length of 3 cycles



(b) The throughput over time for a 10-Burst flow traffic

Figure 4.1: The throughput over time for 1024 flows

N-burst traffic, where each flow receives N consecutive events over N cycles. Figure 4.1a shows the resulting throughput over time for different values of N . We observe that for non-bursty flows, the throughput remains close to one event per cycle. In contrast, under bursty conditions, the throughput drops significantly when we start the system and it still has not received any events (we will refer to this scenario as cold start), but recovers as the number of active flows increases. If we show the total length of the TPE and scheduler pipeline with P , the time it takes to achieve full throughput is the time it takes for there to be at least P active flows in the pipelines. In this plot, the total pipeline length is 8, and we can see that it takes around 800 cycles for a 100-Burst to achieve high throughput. It is important to note that a cold start is very unlikely in the situations we are interested in, such as data centers. However, the design is capable of handling these extreme cases as well.

Pipeline length. Other than the traffic pattern the length of the TPE pipeline can also affect the throughput over time, since with a longer TPE pipeline, it would take longer for a flow to become safe again after leaving the scheduler. Figure 4.1b illustrates the effect of the TPE pipeline length on throughput over time starting from a cold system. Similar to the previous scenario, the throughput is low at the start but increases over time until it eventually becomes one event per cycle. If we show the total length of the TPE and scheduler pipeline with P , the time it takes to achieve full throughput is the time it takes for there to be at least P active flows in the pipelines.

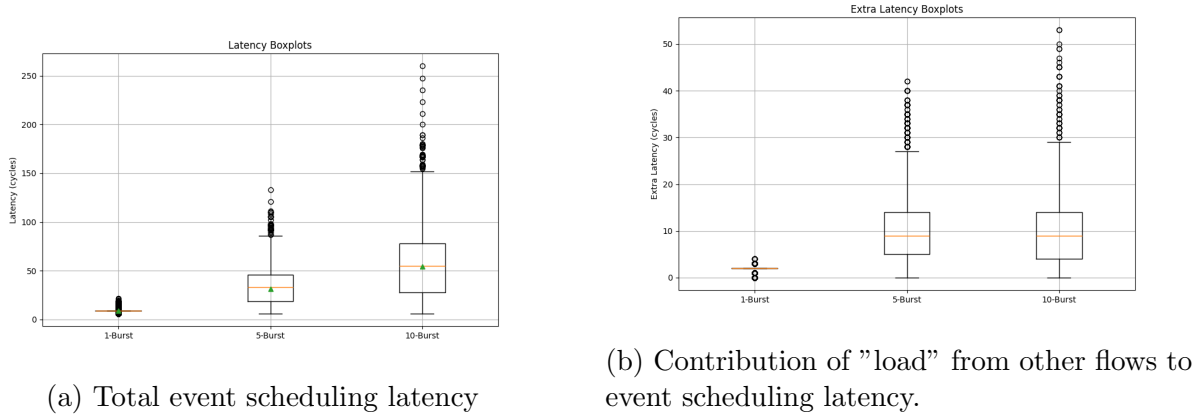


Figure 4.2: Distribution of event scheduling latency for various N-burst traffic patterns for events from 1024 flows over 50 microseconds and a TPE pipeline length of 3.

Event Scheduling Latency

Another important aspect of the scheduler is how long it would take for a particular event to exit the scheduler after its arrival. We compute this value for each event by comparing the time it entered the scheduler pipeline to the time it left it. The latencies for different bursts of data is illustrated in Figure 4.2a. We observe that having bursty traffic increases the event scheduling latency. This is to be expected because the n th event in a burst cannot be processed until all previous events from the same flow have been processed. This would mean that the event would need to wait at least $L + (n - 1) \times P - (n - 1)$ cycles to enter the pipelines, with P being the total length of the TPE and scheduler pipeline and L being the time it takes for the flow ID of the first event to reach the safe flow queues. The reasoning behind this formula is that starting from the arrival of the first event in a burst, it would take at least $L + (n - 1) \times P$ cycles for the $n - 1$ events before the n th event to be processed. But since the n th event arrive $n - 1$ cycles after the first event, we need to deduct this time from the waiting time, resulting in the given formula.

Latency Caused by Other Flows

In the previous section, we mentioned that one of the reasons for the increase of the latency through bursts is that the events in a burst have to wait for the previous events of the same flow to leave the pipeline, which can take some time. In this part, we want to evaluate the latency caused by the presence of other flows in the pipeline, which is the other reason the

latency of a burst may increase. To do this, we evaluate how worse each event of a flow is doing compared to the hypothetical scenario where no other flows were in the scheduler’s pipeline. To do this, we calculate the expected scheduling latency of the i th event of the flow f with the following formula:

$$L_{f,i} = \max(A_{f,i-1} + L_{f,i-1} + P - A_{f,i}, L_{f,0})$$

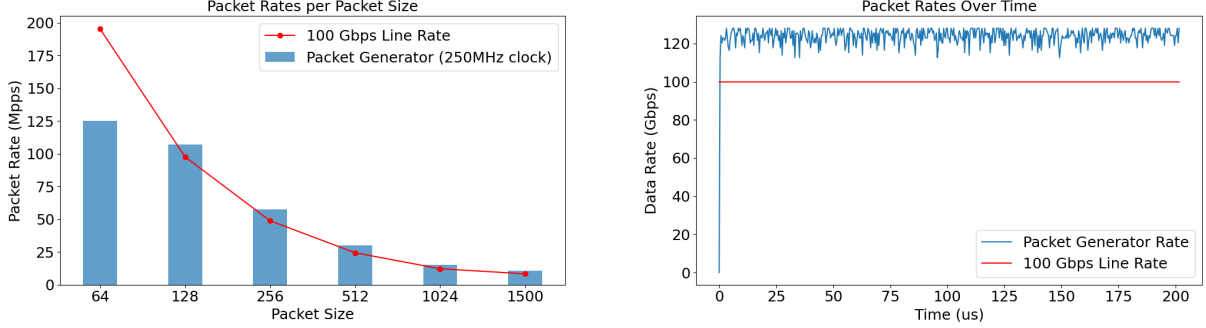
Where $L_{f,i}$ is the scheduling latency of the i th event of flow f , $A_{f,i}$ is the arrival time of the i th event of flow f and P is the total pipeline length of [TPE](#) and scheduler. The scheduling latency is calculated based on the fact that any new event needs to wait for its previous events to finish processing first. Therefore, the earliest an event can leave the pipeline is $A_{f,i-1} + L_{f,i-1} + P$, which is equal to the earliest time the previous event can leave plus the pipeline length. To calculate the scheduling latency, we deduct the arrival time of the i th event from the formula. If this number is less than $L_{f,0}$, this means that by the time this event arrives, the previous one has either left the pipeline or will leave before the event reaches the safe flows. Therefore in this case, the scheduling latency of the event would be the latency of the scheduler’s pipeline from a cold start, which is 5 cycles.

The scheduling latency caused by the presence of other flows, which we will refer to as *Load-Induced Latency*, would be the difference between the actual observed latency and the unloaded latency, which is the latency we computed above. The result can be seen in [Figure 4.2b](#). We observe that, as expected, the increase in the burst does not change the multi-flow induced latency, and the number remains relatively low, as we can observe that in the case of a 10-burst traffic, the median of the latency values is 50 cycles, while the median of the load-induced latency is 10 cycles.

4.2.2 Packet Generator

To evaluate the performance of the packet generator, we verify that it can operate at line rate. We create packet generation requests and feed them as input to the module. It is important to note that the *Fetch Data* stage of the packet generator may take multiple cycles to complete, depending on the packet size. Therefore, the waiting queue for this module may overflow under certain instruction generation rates. In a future version of [HtraP](#), we will add back-pressure support for such scenarios. However, this does not mean that the design cannot handle line rate; it just indicates that events are being generated at a faster rate than they can be processed. In our tests, we generated instructions at a rate that ensured the queues would not overflow.

[Figure 4.3a](#) shows the packet generation rate of the module for different packet sizes. In order to keep up with the 100Gbps line rates, the packet generator would need to



(a) Packet generation rates for packets of different sizes

(b) Packet generation rate over time for a random flow of packet generation instructions

Figure 4.3: Packet generation rates

generate $\frac{10^{11}}{x}$ packets per second, where x is the segment size specified in the instruction. Figure 4.3b shows the packet generation rate of the module under different packet sizes per instruction, where the red line shows the number of packets the model needs to generate to keep up with the 100Gbps line rate. The packet sizes are numbers between 64 bytes to 1500 bytes. The result indicates that with a 250MHz clock, the design would be able to work with a 100Gbps line rate for packets larger than 128 byte. The module cannot keep up with line rate for 64 byte packets. The reason is that the packet constructor module only takes a single cycle to generate the packet, therefore, the packet constructor needs to wait for the update logic to generate the next header before generating the next segment. While for other packet sizes, the next segment information would be already present in the instruction queue by the time the packet constructor is done with the previous segment. 4.3b shows the packet generation rate when the instructions request different packet sizes, where the packet sizes are numbers between 64 bytes to 1500 bytes. We observe that in this scenario the packet generator is able to keep up with 100 Gbps line rate as well.

4.2.3 Reassembly Module

In order for our reassembly module to be effective, it must be able to consume instructions at line rate. This is because packets will arrive at line rate, and therefore events will also be generated at line rate. To prevent the instruction queue in the reassembly module from overflowing, instructions must be processed at the rate they are generated to avoid losing any.

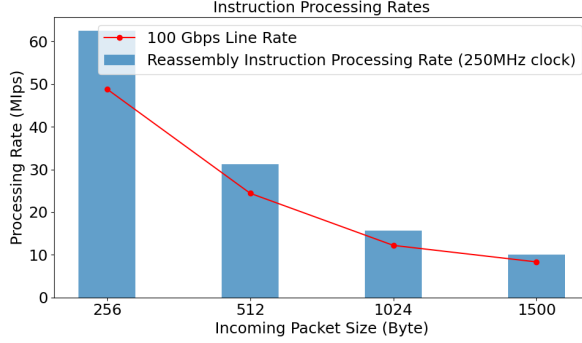


Figure 4.4: Reassembly module’s instruction processing rate for different packet sizes

For the test, we focus specifically on the *add segment* instruction, as it is likely to be generated after each data packet, making it the most frequent reassembly instruction. To evaluate the reassembly module in isolation, we assume that data packets arrive at line rate and generate network events for which the TPE will immediately issue *add data segment* instructions. In reality, there would be some latency before the instruction is generated; however, this assumption does not affect our evaluation, since the latency does not impact the consumption rate.

Figure 4.4 shows the instruction consumption rate for packets of different sizes. The red line indicates the consumption rate required for the reassembly module to keep up with line rate. We observe that the reassembly module can sustain line rate for packets of 256-bytes or larger, but struggles with smaller packet sizes due to the extra cycle introduced by the shift operation, which was explained in Section 2.3.2.

4.3 TPE Programmability

4.3.1 Expressiveness

To evaluate the flexibility and expressiveness of our design, we implemented three distinct transport protocols within the TPE module: TCP, NDP, and RoCEv2. These implementations were developed in HLS and synthesized using the Vitis HLS tool, targeting a 10-nanosecond clock period. The implementation results are presented in Table 4.3. We selected these protocols to represent a broad spectrum of transport layer behaviors. TCP is a traditional stream-based protocol that performs congestion control through packet loss

detection and retransmission timeouts. In contrast, [RoCEv2](#) operates as a message-based protocol and employs rate-limiting to manage congestion. NDP takes a receiver-driven approach, where data transmission is triggered by the arrival of PULL packets. Together, these implementations demonstrate the ability of our architecture to accommodate a variety of transport mechanisms. In this section, we will explain NDP and [RoCEv2](#) and their design considerations. For [TCP](#), a detailed analysis of our implementation can be found in Section 4.4. It is important to note that while we did not configure the [TPS](#) for NDP and [RoCEv2](#), there is enough evidence suggesting that the configuration is possible, which we will also discuss in this section.

Protocol	Implementation (HLS)		Synthesis			
	Event Count	Lines of Code	Latency (cycles)		Resource usage (total)	
			Max	Avg	FF(%)	LUT(%)
TCP	4	612	12	4.50	13384 (< 1%)	7346 (< 1%)
NDP	8	455	3	1.62	7720 (< 1%)	2118 (< 1%)
RoCEv2	19	1434	7	2.82	125101 (< 1%)	39469 (< 1%)

Table 4.3: Protocol implementations in [TPE](#)

NDP

NDP is a receiver-driven, low-latency transport protocol designed for high-throughput datacenter networks. Unlike traditional protocols like TCP, which rely on sender-side congestion control, NDP operates by having receivers explicitly control transmission through PULL packets. In datacenters running NDP, the switches are designed not to drop packets upon congestion, but instead to drop the payload and forward the empty header to the destination. The receiver side of NDP will generate a NACK in response to these headers, and an ACK in response to full packets. In both cases, it will also generate a PULL packet. The PULL packet carries a *pull_counter* field that indicates the total number of pull requests the receiver has sent so far. The pull counter triggers data transmission on the sender side, where the sender keeps track of the *pull_counter* values it has received and ensures that the number of packets it sends after the initial transmission matches the counter. Since the PULL packets are responsible for dictating the flow of data, they are rate-limited by a pre-set amount to control the overall network workload.

To implement NDP, we need to use packet scheduling instructions. As mentioned in section 2.3.1, the current implementation of the packet generator cannot handle these

instructions, however, there is enough evidence to suggest that we can update the module to handle this in a future version.

TPE implementation. To implement the sender side of the NDP, we use a grant based queue in the packet generator. Upon receiving a NACK, the TPE will issue a packet generation instruction for the specific lost packet. After receiving a PULL packet, the sender will issue a token increase instruction with the increase value of $\max(\text{pull_counter}_{\text{current}} - \text{pull_counter}_{\text{prev}}, 0)$ and update the $\text{pull_counter}_{\text{prev}}$ by the same amount. Additionally, it will keep track of the number of lost packets and issue a packet generation command if the number of lost packets is less than the number of available tokens. For the receiver side, we will use a normal queue for ACK and NACK packets and use a rate-limited queue for the PULL packets. Upon receiving either a Data or Header, the receiver will create a packet generation instruction for ACK or NACK, in addition to an instruction for a PULL packet, and send them to the packet generator.

RoCEv2

RoCEv2 is a transport protocol designed to extend the benefits of RDMA over standard IP networks. RDMA allows the applications to perform direct memory operations on a remote endpoint without CPU. In RoCEv2, applications communicate through Queue Pairs(QPs), each consisting of a Send Queue and a Receive Queue. A QP represents a logical communication channel between two endpoints in a connection. Each side can post their requests for sending or receiving data to these QPs. These requests are referred to as Work Queue Element (WQE). The Send Queue holds WQEs describing outbound operations (e.g., sends, writes, reads), while the Receive Queue holds WQEs describing buffers for incoming data. Each WQE specifies details such as the operation type, memory addresses, data length, and control flags for the desired request. Once posted, these WQEs are fetched and executed by the NICs using RDMA, which transfers data directly between local and remote memory, bypassing the host CPU in the data path. Once a WQE is processed, it will be inserted to the Completion Queue.

We implemented DCQCN[44] as the congestion control mechanism for RoCEv2. The DCQCN operates using Congestion Notification Packets (CNPs), which are generated by the receiver. The receiver periodically checks whether any of the packets received since the previous check have their ECN bit set. If so, it generates a CNP. When a CNP is received by the sender, the DCQCN congestion control logic in the NIC updates the sending rate for the affected connection. upon receiving a CNP, it multiplies the current rate by a reduction factor to quickly slow down transmission. This reduced rate limits how fast the QP's WQEs are scheduled for transmission. The QP then enters a recovery phase where

its rate is gradually increased over predefined time intervals until it either reaches its target rate or another CNP is received, in which case the process repeats.

TPE implementation. In our implementation of the [RoCEv2](#) protocol, each QP corresponds to one connection, which means our context memory is responsible for tracking the state of the transport parameters for each QP, and the send and receive queues are stored inside the context. These queues are implemented as ring buffers using C arrays, with one read and one write pointer. This allows read and write operations to be performed efficiently, as they only involve updating the pointers and accessing a specific location in the array. Upon receiving a message, the [TPE](#) adds the metadata to these queues and handles the request based on its type. Each request type (send, receive, read, write and atomic) has its own processor. There are two processors for each request: one processes an event from the application that generates the request on the requester side, and the other handles an event from the network on the receiver side. The latter stores the request in the relevant queue as a [WQE](#) and responds to it. The requester also has processors that handle responses to the sent requests, which may come in the form of data or acknowledgments.

To implement the [DCQCN](#) congestion control mechanism, we use three different timers, two of which are used by the sender to periodically update the rate-related variables according to the DCQCN algorithm. As mentioned earlier, the sender continuously increases the transmission rate until it either reaches a predefined threshold or receives a CNP. The third timer is used by the receiver to generate CNPs if any [ECN](#) marks were detected during the interval.

4.4 Case Study: A TCP implementation in [HtraP](#)

The goal of this case study is to establish that it would be possible to implement a transport protocol in [HtraP](#). To this end, we use the prototype implementation mentioned in Section 4.1, plug-in the TCP logic implemented in [TPE](#), and adjust the configurable parts of the [TPS](#) to be able to execute the [TCP](#) protocol. In this section, we will first discuss the [TPE](#) and [TPS](#) implementations, then connect two copies of the [TCPHtraP](#) modules through a delay line, and simulate the data transfer between the two using a Verilog testbench.

4.4.1 [TPE](#) Implementation

To implement the [TCP](#) protocol, we identified the four different events that direct the actions of the transport layer. These four include:

- **Send Event:** This event is triggered from the application. It asks the protocol to transfer a certain amount of data to the receiver.
- **Data Event:** This event represents the arrival of a data packet. Its fields include the sequence number and the length of the payload.
- **ACK Event:** This event represents the arrival of an acknowledgment. Its fields include the ack sequence and the receiver window size.
- **Timeout Event:** This is a timer generated event that represents a missed acknowledgment.

For each event, we designed an event processor that would carry-out the appropriate action after receiving them. Table 4.4 shows the synthesis report of each individual event processor.

Event Processor	Latency		Resource usage	
	cycles	ns	FF(%)	LUT(%)
Send	2	8.00	2141 (< 1%)	589 (< 1%)
Data	2	8.00	2292 (< 1%)	546 (< 1%)
ACK	12	48.00	6846 (< 1%)	5450 (< 1%)
Timeout	2	8.00	2105 (< 1%)	761 (< 1%)

Table 4.4: The synthesis result for TCP implementation in TPE

Data processor. This event processor would need to keep track of the received packets and generate the correct acknowledgment based on the current state of the window. Ideally, the processor should be able to track the status of each byte in the window, and update these values accordingly. However, due to the limited resources in hardware, we made a slightly simplifying assumption: That the receiver would only be able to handle a limited number of holes in the window. Using this assumption, we represent the window using a small list of ranges, with each element in the list containing a pair of start and end sequence numbers corresponding to the valid data range. After each Data event, the list will be updated according to the sequence number of the arriving packet. If the data creates more holes than allowed, the receiver will drop the packet. For this simulation, we set the size of the list to 2 and implemented the data event processor with this assumption.

4.4.2 TPS Implementation

To configure the [TPS](#) portion of the protocol, we will need to adjust the parameters and configure the flexible modules to handle the logic of the protocol. For the parameters, we used the values presented in [Table 4.2](#). The configuration of each module will be explained in this section.

Packet Generator

As explained in [Section 2.3.1](#), the packet generator has a programmable section called the *header update logic*, which will create the header of the next segment using the current header, instruction information and header update value. In the case of [TCP](#), the field needing an update is the sequence number field. The sequence number of the packet represents the location of its payload in the stream. Therefore, the sequence number of the next segment can be calculated by adding the sequence number of the previous segment to the length of the segment, which is contained in the instruction. We implemented this addition in the packet generation module.

Network and Application Parsers

The parser interfaces have not been implemented in our prototype. For the sake of this case study, we implemented the parsing logic of the application and network parsers, as well as a simplified flow ID generation.

Flow ID mapping. The Flow ID is supposed to be uniquely determined based on four fields: source IP, destination IP, source port and destination port. Since we are testing our design in a test-bench, and not using the actual network, we mapped the Flow ID to the first 10 bits of the source IP of the incoming packet, and assigned the test IPs in a way that would avoid conflict. We believe that this simplification will not hurt the case study, since the effectiveness of using cuckoo hashing for flow ID mapping has already been established in the prior work[\[9\]](#).

Network parser. The two events generated from network events are the data and ACK events. The fields of these events are directly extracted from the packet header. Knowing the exact composition of the header, we wrote a Verilog code that extracts the fields of the event from the header, chooses the correct event type based on the packet length, and sends the generated event to the next stage.

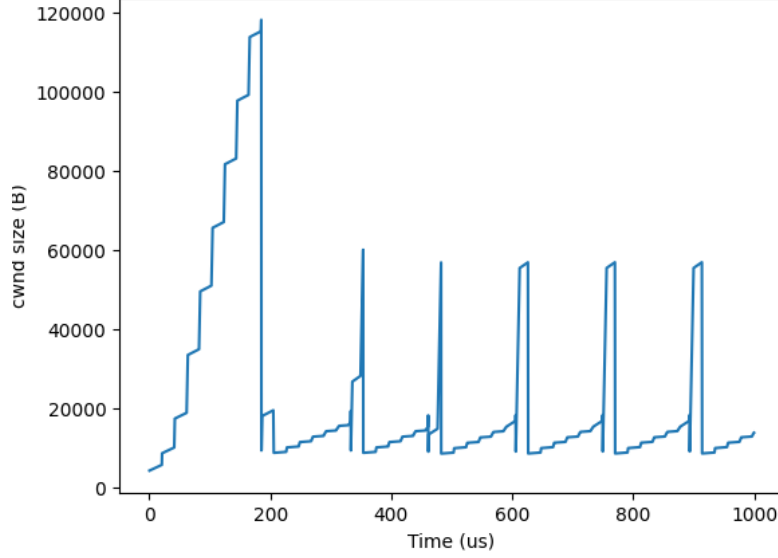


Figure 4.5: The congestion window size over time

Application parser. The only event generated by the application is the send event, with the only field of the event representing the transfer size. In our implementation, this value, along with the flow ID, is generated by [HtraP](#) testbench and provided to the application parser. Therefore, the application parser’s sole responsibility is to generate a *Send event* from this information and forward it to the next stage.

Timer

As mentioned in Section 4.1, the full timer module has not been implemented in this prototype. In order to test the effectiveness of our [TPE](#) implementation, we implemented a timer for the single flow scenario. This timer consists of a counter and a register that keeps the event it is supposed to generate upon expiration. The start command will update the register and reset the counter, the restart command will only reset the counter and the cancel command will stop this counter.

External Memory

The packet generator and reassembly modules are supposed to communicate with the memory to retrieve and store the payload. For the sake of testing, we simulate the behavior of the memory in the testbench using a vector of registers, which will respond to the requests of the two aforementioned modules.

4.4.3 Results

To test our implementation, we will connect two copies of the [TCP HtraP](#) we have implemented. One of them will be the sender and the other will be the receiver. The application input will be generated at the sender and start the transmission process for a single queue, we assume that the connection has already been established. Using this setup, we will test [HtraP](#) with a single flow scenario. The main purpose of this test is to establish that our implementation of [TCP](#) behaves as expected.

Single flow. In this scenario, start the sending process by creating a new application send request for the sender module. Some time after transfer has started, we periodically drop some packets before they can reach the receiver module. [Figure 4.5](#) illustrates the changes in the congestion window size. The flow undergoes the expected slow start, and after a packet loss it detects duplicate ACKs, triggering fast recovery and retransmission with artificial inflation and deflation of the congestion window. Afterward, the plot follows the classic [Additive Increase/Multiplicative Decrease \(AIMD\)](#) algorithm and aligns with the expected behavior of the [TCP](#) protocol.

Chapter 5

Conclusion

5.1 Summary of Contributions

In this thesis, we introduced [HtraP](#), a hardware-programmable framework for the transport layer. [HtraP](#) enables protocol developers to define protocol logic within the [TPE](#), a module that can be programmed using [HLS](#) tools, allowing developers to focus on protocol design without dealing with low-level hardware details. [HtraP](#) receives various transport events, and takes different actions in response to them. The [TPE](#) uses the event, along with information about the current state of the connection, to decide the appropriate actions.

To provide the [TPE](#) with access to connection state information, [HtraP](#) maintains the state of each connection in a per-flow context memory, where each flow corresponds to one connection. The [TPE](#) updates the connection state after processing each event. As a result, no other event belonging to the same connection can enter the [TPE](#) while it is processing an event. A scheduler was designed for [HtraP](#) to ensure that these conditions are met.

The actions requested by the [TPE](#) may include packet generation, data reassembly, and timer manipulation. We designed a separate module to handle each action. The action requests are communicated to these modules through instructions, which are designed to be general enough to support a wide range of transport protocols.

We evaluate [HtraP](#) using a Verilog prototype of the architecture described in [chapter 2](#). Specifically, for packet generation and data reassembly, we implement and optimize these modules to achieve line rate. We forgo the implementation of timer instructions as they have been explored in prior work [\[7\]](#). We assessed the feasibility of the entire [HtraP](#)

hardware, as well as microbenchmarks for the scheduler, packet generator, and reassembly module. To test the effectiveness of the [TPE](#) interface, we implemented and synthesized three different protocols in [HLS](#) for the [TPE](#): [TCP](#), [RoCEv2](#), and [NDP](#). Finally, we fully configured [HtraP](#) for [TCP](#) and tested the implementation with a data transfer from a sender to a receiver.

5.2 Future Work

5.2.1 Supporting Significantly Larger Number of Flows or Per-Flow Context

As seen throughout Chapter 2, our design uses a number of per-flow memories to track various pieces of data related to flows. The scheduler (§2.2.3) keeps the list of event belonging to different flows, the context memory keeps the state of each flow, and the reassembly module (§2.3.2) keeps one or more buffers for each flow. We are currently using [BRAMs](#) to keep all these pieces of data, however, the increase in the width or depth of this memory can pose a challenge due to the limited resources in the hardware. To address this issue, we would need to move some of these contents to the DRAM that is available in the SmartNICs, and use the [BRAMs](#) as cache for the state of the most recently active flows. The DRAM memory is slower than [BRAM](#), but can allow us to store more data. We need to design a module that would decide which flow should move to the DRAM. An example of such moving logic has been used by F4T[9], where it swaps-out the least recently active flow to the DRAM and bring in another flow.

5.2.2 Enabling Back-Pressure for The Packet Generator

As mentioned in Section 4.2.2, packet generation requests may fill up the queues if they are produced faster than they can be consumed. To address this issue, a back-pressure mechanism would need to be introduced into the packet generator. This mechanism would allow the packet generator to prevent previous modules from generating additional requests until it is ready to accept more.

One way to implement back-pressure is to allow the packet generator to stall the [TPE](#) and stop the scheduler from dequeuing more events. However, this approach may cause head-of-line blocking for other event types that do not require access to the packet generator. To avoid this, we can implement a mechanism that allows the packet generator

to selectively stall only a subset of events. Namely, the event types that would lead to packet generation. Our current design does not support this, as the scheduler does not differentiate between event types. A future version of the [HtraP](#) scheduler could address this limitation by tracking event types and avoiding the dispatch of certain types when back-pressure is active.

5.2.3 Higher level programming framework

One of our goals in designing [HtraP](#) has been to make programming easier and allow programmers to implement protocols without concerning themselves with low-level details. We achieved this by limiting most of the programming to the [HLS](#)-based [TPE](#) module and providing [TPE](#) with an interface to instruct the other modules to carry out the tasks needed for the protocol. We can take this goal one step further by proposing a higher-level programming framework for [TPE](#). To establish the motivation, we consider two situations in our design:

1. As discussed in Section [3.1.2](#), there are design choices that the programmer must handle carefully, such as loop or array implementations. While these concerns are at a higher level than the timing and resource issues typically associated with hardware programming, they are still relatively low-level and ideally should be abstracted away from the programmer.
2. In [HtraP](#), the implementation of the packet generator’s header update logic, as well as the parsers, must be done separately, even though they are tightly coupled with the [TPE](#). It would be more convenient if the logic for all programmable components could be specified in a single program, and later mapped by a compiler to the appropriate modules.

We conclude that a higher-level programming framework can provide better abstraction and further simplify the design process. One potential candidate is MTP[[32](#)], a transport-layer programming framework that enables protocol designers to specify high-level protocol behavior using a domain-specific language and then compile it to various hardware targets. [HtraP](#) aligns well with the abstractions provided by MTP and could potentially serve as a backend for MTP programs, where MTP code could be compiled to fill the [TPE](#), header update logic, and parser logic.

References

- [1] AMD OpenNIC Project. <https://github.com/Xilinx/open-nic>. Accessed: June 2025.
- [2] AMD Vitis HLS. <https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html>. Accessed: June 2025.
- [3] RDMA - iWARP. <https://www.chelsio.com/nic/rdma-iwarp/>. Accessed: June 2025.
- [4] RoCE Introduction. <https://www.roceinitiative.org/roce-introduction/>. Accessed: June 2025.
- [5] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center tcp (dctcp). In *Proceedings of the ACM SIGCOMM 2010 Conference*, pages 63–74, 2010.
- [6] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is more: Trading a little bandwidth for Ultra-Low latency in the data center. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 253–266, San Jose, CA, April 2012. USENIX Association.
- [7] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 93–109, Santa Clara, CA, February 2020. USENIX Association.
- [8] Serhat Arslan, Stephen Ibanez, Alex Mallery, Changhoon Kim, and Nick McKeown. Nanotransport: A low-latency, programmable transport layer for nics. In *Proceedings of the ACM SIGCOMM Symposium on SDN Research (SOSR)*, SOSR ’21, page 13–26, New York, NY, USA, 2021. Association for Computing Machinery.

- [9] Junehyuk Boo, Yujin Chung, Eunjin Baek, Seongmin Na, Changsu Kim, and Jangwoo Kim. F4t: A fast and flexible fpga-based full-stack tcp acceleration framework. In *Proceedings of the 50th Annual International Symposium on Computer Architecture, ISCA '23*, New York, NY, USA, 2023. Association for Computing Machinery.
- [10] Qizhe Cai, Mina Tahmasbi Arashloo, and Rachit Agarwal. dcvim: Near-optimal proactive datacenter transport. In *Proceedings of the ACM SIGCOMM 2022 Conference*, pages 53–65, 2022.
- [11] V. Cerf and R. Kahn. A protocol for packet network intercommunication. *IEEE Transactions on Communications*, 22(5):637–648, 1974.
- [12] Mo Dong, Qingxi Li, Doron Zarchy, P. Brighten Godfrey, and Michael Schapira. PCC: Re-architecting congestion control for consistent high performance. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*, pages 395–408, Oakland, CA, May 2015. USENIX Association.
- [13] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 51–66, Renton, WA, April 2018. USENIX Association.
- [14] Peter X Gao, Akshay Narayan, Gautam Kumar, Rachit Agarwal, Sylvia Ratnasamy, and Scott Shenker. phost: Distributed near-optimal datacenter transport over commodity network fabric. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, pages 1–12, 2015.
- [15] Chuanxiong Guo, Haitao Wu, Zhong Deng, Gaurav Soni, Jianxi Ye, Jitu Padhye, and Marina Lipshteyn. Rdma over commodity ethernet at scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 202–215, 2016.
- [16] Sangtae Ha, Injong Rhee, and Lisong Xu. Cubic: a new tcp-friendly high-speed tcp variant. *ACM SIGOPS operating systems review*, 42(5):64–74, 2008.

- [17] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W Moore, Gianni Antichi, and Marcin Wójcik. Re-architecting datacenter networks and stacks for low latency and high performance. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 29–42, 2017.
- [18] Zhenhao He, Dario Korolija, and Gustavo Alonso. Easynet: 100 gbps network for hls. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 197–203, 2021.
- [19] EunYoung Jeong, Shinae Wood, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mTCP: a highly scalable user-level TCP stack for multicore systems. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 489–502, Seattle, WA, April 2014. USENIX Association.
- [20] Cheng Jin, David X Wei, and Steven H Low. Fast tcp: motivation, architecture, algorithms, performance. In *IEEE INFOCOM 2004*, volume 4, pages 2490–2501. IEEE, 2004.
- [21] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. Tas: Tcp acceleration as an os service. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys ’19, New York, NY, USA, 2019. Association for Computing Machinery.
- [22] Gautam Kumar, Nandita Dukkupati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. Swift: Delay is simple and effective for congestion control in the datacenter. 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*, SIGCOMM ’20, page 514–528, New York, NY, USA, 2020. Association for Computing Machinery.
- [23] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasnic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *Proceedings of the conference of the ACM special interest group on data communication*, pages 183–196, 2017.
- [24] Bojie Li, Kun Tan, Layong (Larry) Luo, Yanqing Peng, Renqian Luo, Ningyi Xu, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Clicknp: Highly flexible and high

- performance network processing with reconfigurable hardware. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 1–14, New York, NY, USA, 2016. Association for Computing Machinery.
- [25] Yuliang Li, Rui Miao, Hongqiang Harry Liu, Yan Zhuang, Fei Feng, Lingbo Tang, Zheng Cao, Ming Zhang, Frank Kelly, Mohammad Alizadeh, and Minlan Yu. Hpec: high precision congestion control. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 44–58, New York, NY, USA, 2019. Association for Computing Machinery.
 - [26] Yuanwei Lu, Guo Chen, Bojie Li, Kun Tan, Yongqiang Xiong, Peng Cheng, Jiansong Zhang, Enhong Chen, and Thomas Moscibroda. Multi-Path transport for RDMA in datacenters. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 357–371, Renton, WA, April 2018. USENIX Association.
 - [27] Yuanwei Lu, Guo Chen, Zhenyuan Ruan, Wencong Xiao, Bojie Li, Jiansong Zhang, Yongqiang Xiong, Peng Cheng, and Enhong Chen. Memory efficient loss recovery for hardware-based transport in datacenter. In *Proceedings of the First Asia-Pacific Workshop on Networking*, APNet '17, page 22–28, New York, NY, USA, 2017. Association for Computing Machinery.
 - [28] Ilias Marinos, Robert N. M. Watson, and Mark Handley. Network stack specialization for performance. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, New York, NY, USA, 2013. Association for Computing Machinery.
 - [29] Radhika Mittal, Vinh The Lam, Nandita Dukkkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. *SIGCOMM Comput. Commun. Rev.*, 45(4):537–550, August 2015.
 - [30] Radhika Mittal, Vinh The Lam, Nandita Dukkkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. Timely: Rtt-based congestion control for the datacenter. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, page 537–550, New York, NY, USA, 2015. Association for Computing Machinery.
 - [31] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for rdma. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data*

- Communication*, SIGCOMM '18, page 313–326, New York, NY, USA, 2018. Association for Computing Machinery.
- [32] Pedro Vitor Valenca Mizuno. *Exploring high-level transport programming through eXpress Data Path*. Masters thesis, University of Waterloo, January 2025. Available at <https://hdl.handle.net/10012/21386>.
 - [33] Behnam Montazeri, Yilong Li, Mohammad Alizadeh, and John Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 221–235, 2018.
 - [34] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. AccelTCP: Accelerating network applications with stateful TCP offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 77–92, Santa Clara, CA, February 2020. USENIX Association.
 - [35] Ali Munir, Ihsan A. Qazi, Zartash A. Uzmi, Aisha Mushtaq, Saad N. Ismail, M. Safdar Iqbal, and Basma Khan. Minimizing flow completion times in data centers. In *2013 Proceedings IEEE INFOCOM*, pages 2157–2165, 2013.
 - [36] Mario Ruiz, David Sidler, Gustavo Sutter, Gustavo Alonso, and Sergio López-Buedo. Limago: An fpga-based open-source 100 gbe tcp/ip stack. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 286–292, 2019.
 - [37] Ahmed Saeed, Nandita Dukkupati, Vytautas Valancius, Vinh The Lam, Carlo Contavalli, and Amin Vahdat. Carousel: Scalable traffic shaping at end hosts. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17*, page 404–417, New York, NY, USA, 2017. Association for Computing Machinery.
 - [38] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 87–102, Renton, WA, April 2022. USENIX Association.
 - [39] Kun Tan, Jingmin Song, Qian Zhang, and Murad Sridharan. A compound tcp approach for high-speed and long distance networks. In *Proceedings-IEEE INFOCOM*, 2006.

- [40] Balajee Vamanan, Jahangir Hasan, and TN Vijaykumar. Deadline-aware datacenter tcp (d2tcp). *ACM SIGCOMM Computer Communication Review*, 42(4):115–126, 2012.
- [41] Keith Winstein, Anirudh Sivaraman, and Hari Balakrishnan. Stochastic forecasts achieve high throughput and low delay over cellular networks. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 459–471, Lombard, IL, April 2013. USENIX Association.
- [42] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. Ictcp: Incast congestion control for tcp in data-center networks. *IEEE/ACM Transactions on Networking*, 21(2):345–358, 2013.
- [43] Francis Y. Yan, Jestin Ma, Greg D. Hill, Deepti Raghavan, Riad S. Wahby, Philip Levis, and Keith Winstein. Pantheon: the training ground for internet congestion-control research. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 731–743, Boston, MA, July 2018. USENIX Association.
- [44] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication, SIGCOMM ’15*, page 523–536, New York, NY, USA, 2015. Association for Computing Machinery.
- [45] Yibo Zhu, Haggai Eran, Daniel Firestone, Chuanxiong Guo, Marina Lipshteyn, Yehonatan Liron, Jitendra Padhye, Shachar Raindel, Mohamad Haj Yahia, and Ming Zhang. Congestion control for large-scale rdma deployments. *SIGCOMM Comput. Commun. Rev.*, 45(4):523–536, August 2015.